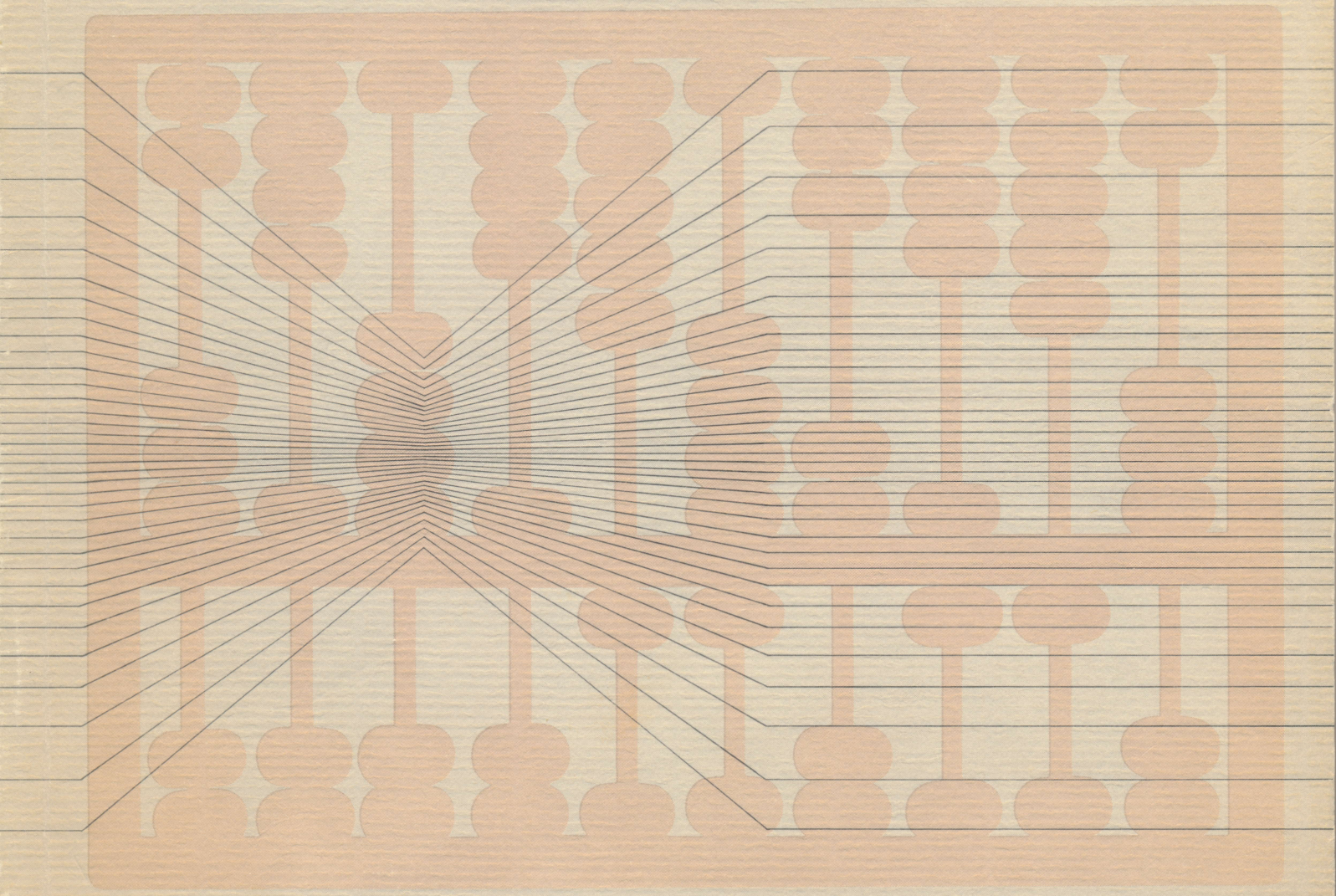
I. P. Sharp Associates

SHARP APL
Report Formatting



SHARP APL REPORT FORMATTING

This manual is a revised version of **SHARP APL Report Formatting**, publication 220SA06, 1977, which in turn superseded **SHARP APL Report Formatting**, publication 220SA05, 1975. It includes material described more concisely in **SATN-30, Numeric Display**, 1979, **SATN-17, Formatting Primitive**, 1976, and **SATN-18, ⎕*FMT***, 1976.

Revised August 1979.

Printed in Canada.
Publication Code: 220SA07
ISBN 0 86493 028 3

CONTENTS

		Page
1.	FORMATTING DATA TABLES WITH ⍕ AND ⎕*FMT*	1
	What the format primitives are for	1
	Preview summary of two formatting primitives	2
	Types of ⎕*FMT* formatting phrases	2
	List of ⎕*FMT* formatting phrases	3
	Special property of ⎕*FMT* displays and available work area	3
	Special property of ⎕*FMT* when writing ⎕*OUT* files	3
2.	SOME GENERAL POINTS ABOUT FORMATTING THAT APPLY BOTH TO ⍕ AND TO ⎕*FMT*	5
	A note about "columns" and "fields"	5
	Predictable size and shape	5
	Field too small to show the numbers in the format you've specified	6
	Field wider than needed to show the data	6
	More digits requested than system can supply	7
3.	DETAILED DESCRIPTION OF ⍕	8
	Capturing the way the system displays numbers	8
	With monadic thorn, significant digits are controlled by ⎕*PP*	8
	Width of fields in result of monadic ⍕	9
	The result generated by monadic thorn is not affected by ⎕*PW*	9
	Monadic thorn applied to a character array	10
	Dyadic thorn: controlling field width and precision	10
	Fixed-point decimal format with dyadic thorn	11
	Selecting decimal or exponential form	11
	Number of control pairs required by dyadic thorn	11

One control pair applied to all columns 12

Field width 0 lets the system pick the width 12

Space for the negative sign 13

A single system-picked width for all columns 13

Leading and trailing zeros with thorn 13

Picking an appropriate width for decimal displays 14

Representing numbers in exponential form 15

Exponential format with dyadic thorn 16

System-picked width in exponential form 17

Minimum field width for exponential form 17

Decimal and exponential fields in the same display 17

Insufficient space for the result in exponential forms 18

More digits than the system can supply 18

4. DETAILED DESCRIPTION OF ⎕*FMT* 19

The right argument: the data to be formatted 19

Shape of the arrays to be formatted 19

Formatting several arrays when some require more rows than others 20

The left argument: the format control vector 20

Separating one phrase from the next 21

Repeated phrases 21

Literal text within a format phrase 22

The structure of a format phrase 23

The "right argument" of a format phrase 23

Types of "left argument" to a format phrase 24

List of ⎕*FMT* qualifiers and decorators 24

Qualifiers and decorators and the formats with which they are available 25

Format by picture 25

Decimal point in picture format 26

Leading and trailing zeros in picture format 27

Position of negative sign or decorators in picture format 27

Sign-dependent decorators with picture format 28

Floating dollar sign in picture format 28

Trailing zeros and blanked-out field for zero value 29

Field overflow with picture format 29

Background text to fill `Z` positions left blank 30

Substituting other characters for `9` or `Z` in the picture, or for `0` or `*` in the result 31

The integer format *I* and fixed-point format *F* 32

Exponential form with □*FMT* format *E* 33

Alignment of columns 35

Qualifiers usable with *I*, *F* and *E* formats 35

Scale factor *K* 36

Left justifying with qualifier *L* 36

Blank for zero value with *B* 37

Leading zeros with *Z* 37

Insertion of commas with *C* 38

Substitutes for standard characters in *I*, *F* and *E* formats 39

Using substitute characters to insert dividing markers 41

The sign-dependent decorators *M N P Q* 42

Decorator on both sides of the number 43

Floating dollar sign 43

Deliberately aligning negative values differently 44

Leading decorators combined with leading zeros 44

Sign-dependent decorators with *E* format 44

Background text with *R* 45

Background text to mark zero values 46

Formatting characters with ⎕*FMT* code *A* 47

Using *A* format to display a vector 48

Background text in the spaces between characters 50

Other qualifiers or decorators with *A* format 50

Data of the opposite type 50

Inserting a constant text 51

Constant text that appears after the last column of data 51

Empty fields 52

Moving to a new position in the display with phrases *X* and *T* 53

Moving back over the fields already written 54

Overwriting one field with another 55

Skipping to the end of the positions already formatted 56

APPENDIX: CONDENSED SUMMARIES OF FORMATTING RULES 57

1. Shape and conformability 57

Monadic ⍕ 57

Dyadic ⍕ 57

Dyadic ⎕*FMT* 57

2. Structure and meaning of control pairs in the left argument of ⍕ 58

Structure 58

Meaning 58

3. Structure of ⎕*FMT* control vector 59

Literal text 59

Blanks 59

Separation into phrases 59

Repetition of phrases 59

Last phrase used 59

Type of data 60

4. Components of □*FMT* phrase 60

5. Effects of □*FMT* qualifiers 60

K: Scale factor 60

B: Blank zero values 61

Z: Leading zeros 61

C: Commas 61

L: Left justify 61

6. Effects of □*FMT* decorators 62

P and *M*: Leading sign-dependent decorators 62

Q and *N*: Trailing sign-dependent decorators 62

R: Background text 63

S: Substitution of standard characters 63

INDEX 65

CHAPTER 1

FORMATTING DATA TABLES WITH ⍕ AND ⎕FMT

What the format primitives are for

Once your program has completed its calculations, probably what you want next is to print a report. If all you need is to display the values stored in a numeric matrix, for example, one called *X*, then the statement

```
X
```

causes the system to print that table. If you don't give any explicit instruction about **how** to display *X*, the system falls back on its own rules about the appearance and spacing of the display.

But what if you want to go beyond that? Perhaps you want to display the numbers in a particular way, or to insert (between the columns of figures) character information, such as captions or labels. Perhaps you need commas to separate the thousands from the millions, or a floating dollar sign on each row; or perhaps you need to mark the negative values with the message "OVERDRAFT" instead of the high minus sign that APL otherwise provides. That's when you need one of the formatting primitives. There are two of them:

⍕ Called "format" (because of what it does) or "thorn" (because of the way the symbol looks)

⎕*FMT* Called "report formatter" or simply "quad FMT"

Each of these can produce a character array from a numeric argument. Of the two, ⍕ is the simpler; it just takes one numeric array, and produces the corresponding character array. You can if you wish control the number of decimals and the spacing of each column of numbers.

⎕*FMT* goes much further; it lets you combine several different arrays into a single table. It also lets you insert character information, and gives you detailed control of the way in which numeric quantities are represented.

Preview summary of the two formatting primitives

`⍕ X`	Gives you the array of characters that the system would use to display *X* if you had simply typed *X*.
F `⍕ X`	Gives you an array of characters representing *X*. For each column of *X*, the left argument *F* lets you specify (1) the total width of the field; (2) whether numbers are represented in decimal or in exponential form; and (3) the number of digits after the decimal point (for a column represented in decimal form), or the number of significant digits (for a column represented in exponential form).
F `⎕FMT X` *F* `⎕FMT (X;Y;+/Z*2)`	Gives you a matrix of characters representing the columns of *X*, or of the various arrays listed between parentheses and separated by semicolons. An array may be the result of an expression (as illustrated here by `+/Z*2`, the sum of the squares of *Z*).

Types of *⎕FMT* formatting phrases

The left argument of *⎕FMT* is a character vector, made up of **phrases**, separated from each other by commas. Each phrase controls the way a column of the right argument is represented in the result, or else inserts additional text or spaces into the result.

There are eight types of format phrase. In general, a phrase's type is indicated by a key letter contained within it, usually the rightmost letter in the phrase. One phrase, used solely to insert a fixed text, has no letter, but is distinguished by being completely enclosed by delimiting symbols.

List of □*FMT* Formatting Phrases

G	**Picture**	Treats a column of numbers as integers, and inserts the digits that represent them into a pattern provided as part of the phrase.
I	**Integer**	Represents a column of numbers to the nearest whole number (integer), and controls the width in which they appear.
F	**Fixed point**	Represents a column of numbers as fixed-point decimals, and controls the width and the number of decimal places.
E	**Exponential**	Represents a column of numbers in exponential form, and controls the width and the number of significant digits.
A	**Character data**	Controls the width (that is, the number of print positions) used to display a column of characters.
T	**Absolute reposition**	Before starting to format the next field, moves to the absolute print position shown.
X	**Relative reposition**	Before starting to format the next field, moves forward or back the indicated number of print positions.
<TEXT>	**Text insertion**	Inserts the text between the delimiters < and > on each row of the result, before formatting the next column. (Certain other delimiters are also permitted, if the ones shown aren't convenient.)

The *I*, *F*, and *E* phrases can be used with a number of **qualifiers** that do such things as print or suppress leading zeros, insert commas, or make blank the representation of a number whose value is zero. The *G*, *I*, *F*, and *E* phrases can also be combined with **decorators**, which permit you to specify how negative or positive values are to be distinguished, and to fill in the unused portions of the field. These are all explained in detail in later sections.

Special property of □*FMT* displays and available work area

If your program uses □*FMT* to generate displays directly (rather than using □*FMT* to calculate the value of a variable), the system prepares and prints the output one line at a time. That way, it is not necessary to prepare the entire display within the workspace before printing it. As a consequence, a display can be printed even though it is so large that there would not be room in the workspace to contain all of it at once.

Special property of □*FMT* when writing □*OUT* files

By using □*OUT*, you can specify that output (otherwise directed to your terminal) should be appended to a file. (For details concerning □*OUT*, see SATN-3.) If you use that option, you can at the same time benefit from the following special property of □*FMT*.

If the output is sent to a □*OUT* file **and** is evoked **directly** by a □*FMT* statement (rather than being first stored as the value of a variable), **execution of the □*FMT* statement is deferred.**

The components written to the file contain first a code meaning that what follows is the record of a □*FMT* transaction, then the format control vector that you used, and then the actual data to be formatted. The formatting is **not** done at that time.

One of the main reasons for diverting your output to a □*OUT* file is that you expect later on to use *HSPRINT* to have the file printed. (*HSPRINT* is described in SATN-8.) If the file is printed by *HSPRINT*, the formatting is carried out then.

The practical consequence of deferring the formatting until later is that it is then done not by your terminal task, but by the background job which does the printing. The cost of printing a large volume of report using *HSPRINT* is significantly lower than printing it at a terminal, and often it is more convenient.

CHAPTER 2

SOME GENERAL POINTS ABOUT FORMATTING THAT APPLY BOTH TO ⍕ AND TO ⎕FMT

A note about "columns" and "fields"

The formatting functions let you control the way each column of a table of numbers is displayed. In talking about this process, it's useful to distinguish between **columns** and **fields.** (These aren't really technical terms with agreed-on meanings in data processing, so the way they're distinguished in this manual may not help you anywhere else.)

A **column** is what you get by selecting a particular position along the last axis of an array of data. Almost all formatting is concerned with representing numbers. A particular column of numbers may contain big numbers or small numbers. Some numbers may require more characters to represent them than others. Nevertheless, they're all part of the same column. The following is a table with three columns of numbers (shown the way the system would display them when you don't give explicit formatting instructions):

```
1960    1.03        14005.6
1961    1.075       18763
1962    1.06        19427.885
1963    1.086666667 22600
```

A **field** is a vertical band of one or more consecutive print positions (or, if you like, columns of characters) used for the same purpose. Generally speaking, you have a **field** in the result in which to represent each **column** of the argument. For example, the shading below shows a possible display of the same numbers, divided into three fields. The shading isn't part of the display, but is overlaid to indicate where one field ends and the next begins.

```
FY 1960  1.03∘/∘   $14,006
FY 1961  1.075∘/∘  $18,763
FY 1962  1.06∘/∘   $19,428
FY 1963  1.087∘/∘  $22,600
```

The formatting functions control the width of the **fields** that are used to represent the **columns** of numbers in the argument. The first column of the table illustrated above could be written in a field 4 characters wide. So could the second column, provided you were willing to round off some of the fractional parts. The third column would require a field 5 characters wide, to contain the 5 digits, and of course wider if you also want commas, dollar signs, etc.

In displaying a table such as that one, you almost certainly want to have enough blank space that the digits from adjacent fields don't run together. That is usually done by specifying a value for the field width that is at least one greater than the width needed for the widest representation you expect to print.

Predictable Size and Shape

In general, the shape of an array produced by formatting depends on the left argument and the number of columns in the right argument, but not on the particular values occurring in those columns.

That's true without exception for ⎕*FMT*, and with two exceptions for ⍕: (1) the thorn function can be used with **no** left argument; and (2) within the left argument of ⍕, you have the option of using the number 0 where you would otherwise specify the width of the field. In those cases, the system figures out for itself how many characters wide a field should be.

In all other uses of ⍕, and in **every** use of ⎕*FMT*, the width of each field is controlled by what you put in the left argument.

Field too small to show the numbers in the format you've specified

Suppose a number can't be written in the number of characters you've allowed for it. If you haven't allowed enough space in which to write the representation of a number, you have a **field overflow.** All of the positions devoted to that number are filled with stars. That way you don't see **any** of its digits, but you know that your data contains a number which can't be written in the space provided. For example, suppose you try to write the numbers 999, 1000, and 998 in a field only three print positions wide. You get:

```
999
***
998
```

You must also allow space for such decorators as are routinely supplied by the system (for example, the negative sign) or as you may have optionally specified. Lack of space in which to write the decorators also causes field overflow. In the same three print position format, trying to write the numbers ¯1 ¯10 ¯100 has the following result:

```
 ¯1
¯10
***
```

Field wider than needed to show the data

Fields are always filled from the right (unless you explicitly direct otherwise by using the "left justify" option with ⎕*FMT* formats *I*, *F*, or *E*). If you have specified a total width that is wider than needed to represent your data (including the space required for optional text or decorators) the remaining width is filled with blanks on the left side. (There's also an option with the ⎕*FMT* formats *G*, *I*, *F*, and *E* to supply your own background text to appear in the unused print positions, rather than blanks.)

Suppose a data table has three columns, each of them containing the four numbers 1, 23, 456, and 7890. You can see below how those appear if they are formatted in fields four, five, and nine print positions wide:

Field width 4–

```
   1   1   1
  23  23  23
 456 456 456
7890789078907890
```

Field width 5—

```
    1    1    1
   23   23   23
  456  456  456
 7890 7890 7890
```

Field width 9————

```
        1        1        1
       23       23       23
      456      456      456
     7890     7890     7890
```

Notice that it is usually desirable to have a certain number of blanks at the left of each field, in order to separate adjacent columns and to space your display for a pleasing appearance.

More digits requested than the system can supply

The SHARP APL system represents numbers internally to a precision equivalent to about eighteen decimal digits. If you ask to have a number represented with more precision than the system can supply, it gives you as many of the digits as it can, but writes 0 for any position further to the right than the eighteenth significant digit.

CHAPTER 3

DETAILED DESCRIPTION OF ⍕

Capturing the way the system displays numbers

The function ⍕ takes as its argument a numeric array. It may be used either monadically (that is, without a left argument) or dyadically (with a left argument).

The **monadic format** gives you a character array containing the characters that the system would use (if not instructed more specifically) to display that array. Suppose you have a numeric array called *X*. Then

```
      ⍕X
```

gives you exactly the same characters that you would see at the terminal if you had simply typed

```
      X
```

To the eye, the effect of the statement ⍕*X* is just the same as *X*. (Well, almost the same. They look exactly the same so long as they fit within the printing width at your terminal. They **might** look different if the display is too wide to fit within the printing width. Display of a numeric array evoked by simply writing *X* is broken at a space, so that no number's representation appears partly on one line and partly on the next. The system doesn't do that for character displays, and hence doesn't do it for the result of ⍕*X*.)

The useful effect of formatting a numeric array is that, since the result is a character array, you can now join to it other character arrays in order to produce a formatted table, or you can store the resulting array of characters as the value of a variable.

With monadic thorn, significant digits are controlled by ⎕*PP*

When the system displays numbers, the number of significant digits printed is controlled by the system variable ⎕*PP* (for "printing precision"). The result of ⍕*X* is affected in exactly the same way. For example, suppose *X* is the 3-by-4 matrix formed by *X*←3 4⍴÷⍳6. Then the display of *X* and of ⍕*X* look just alike:

```
      ⎕PP
10

      ⍕X
1             0.5           0.3333333333 0.25
0.2           0.1666666667  1            0.5
0.3333333333  0.25          0.2          0.1666666667
```

```
      ⎕PP←4

      ⍕X
1       0.5     0.3333 0.25
0.2     0.1667  1      0.5
0.3333  0.25    0.2    0.1667
```

Width of fields in result of monadic ⍕

The result of formatting a vector is a vector with one blank separating the representation of each element. Each field has its own width; that is, the number of print positions devoted to any element is independent of the print positions required for other elements.

There's one blank separating adjacent fields. Thus, the number of blanks in the representation of a vector is always 1 less than the number of elements in the vector. But that's true only for vectors. An array having two or more axes is formatted by columns. There's a field for each column of the array being formatted. All the fields have the **same** number of print positions available for printing. In addition, adjacent fields are separated by one additional column that is always entirely blank (so that the number of extra blank positions is 1 less than the number of columns in the array).

Not counting the separator columns, the maximum number of positions available for printing is ⎕*PP*+6. But if the system can format the entire array by using fewer than ⎕*PP*+6 positions throughout, it does.

Stating the same thing a different way, you could imagine that the system finds **w**, the width needed to represent all the elements as a single column, and then uses **w** as the width of the first field, and **w**+1 as the width of each of the others.

The result generated by monadic thorn is not affected by ⎕*PW*

The printing **width** ⎕*PW* does not affect the result produced by monadic format (even though, as noted above, ⎕*PW* may affect the way it appears at the terminal when you display it, just as it affects **all** output to the terminal). If you have a matrix *Y* that contains twenty separate columns of fractions, then the result of converting that array to characters may be far too wide to fit in the width of a normal terminal display. But that doesn't affect the number of print positions it requires. For example:

```
      Y←3 20 ⍴ ÷⍳60

      ⍴ ⍕Y
3 279
```

That shows that, when ⎕*PP* is 10, it takes 279 print positions to format those 20 columns of numbers. (They aren't shown here because it's impractical to show a matrix of 279 print positions on page of this size.) If you should ask to have that formatted array displayed at the terminal, the long lines of output would still be broken up to fit in the display area, under the control of ⎕*PW*, as usual.

Monadic thorn applied to a character array

If you write

```
⍕X
```

and X is a character array already, the result is the same character array unchanged.

Dyadic thorn: controlling field width and precision

When ⍕ is used with a left argument, the left argument controls the width and the number of decimal places of each field in the result.

The left argument should be a numeric vector (or a scalar). It is treated as a sequence of **pairs** of numbers, one pair for each column of the right argument. For example, to control the display of a table having 3 columns, the left argument of ⍕ must have 6 elements.

Within each pair, the **first** number gives the width of the field in which that column is to be represented. The first number must therefore be a positive integer.

Suppose X is the 3-by-3 matrix `X←⌹3 3⍴÷⍳9`. Here it is displayed without format instructions:

```
1            0.25          0.1428571429
0.5          0.2           0.125
0.3333333333 0.1666666667 0.1111111111
```

Suppose you have the following requirements for the display of X:

Field 1: 5 characters wide, no decimals.

Field 2: 8 characters wide, 3 decimals.

Field 3: 5 characters wide, 2 decimals.

You can get that by the following:

```
      5 0  8 3  5 2 ⍕ X
1   0.250 0.14
1   0.200 0.13
0   0.167 0.11
```

(Notice that numbers whose representations must be truncated are rounded.)

Fixed-point decimal format with dyadic thorn

If the number of decimal places you specify is 0, you get your display rounded to the nearest integer, and without a decimal point.

If the number of decimal places is greater than 0, you get a decimal point, and that many places of decimals, including trailing zeros.

A negative value is marked by the high minus sign ¯ appearing to the left of the leftmost digit. There is no provision with ⍕ to mark negative values in any other way. The total field width must be great enough to allow room to print the negative sign. Trailing zeros are supplied, and a zero in the units position, but no other leading zeros.

Selecting decimal or exponential form

The **second** member of each pair controls the number of decimal places used in the display of that column, and also whether the format is fixed-point or exponential. The convention is this:

2nd element **positive:** Fixed-point decimal form, with that many places to the right of the decimal point.

2nd element **negative:** Exponential form. The number of significant digits displayed is the absolute value of the second element.

The form selected applies to a particular column, so, although any one column must appear in the same format throughout, it's quite possible to have one column of an array in decimal form and another in exponential form. In the sections that follow, first the ⍕ decimal form is described, and then the ⍕ exponential form.

Number of control pairs required by dyadic thorn

The length of the left argument of dyadic ⍕ must meet one of the following conditions:

1. One control pair for each column in the right argument. That is, the left argument is a vector whose length is twice the length of the last axis of the array being formatted. All the preceding examples met that criterion.

2. The number of control pairs on the left goes evenly into the number of columns on the right. In that case, the vector left argument is repeated cyclically as many times as necessary to describe the columns on the right.

   Perhaps the simplest instance of this is a left argument consisting of a single control pair (that is, a vector of length 2). In that case, that pair applies to **all** the columns of the right argument, however many there are.

3. There's just one number (a scalar or a 1-element vector). That is treated in the same way as a single pair, with the first element of the pair being zero.

Any other length for the left argument of ⍕ is rejected as a *LENGTH ERROR*. (The rules for the number of pairs in the left argument of ⍕ are stricter than the rules for the number of phrases in the left argument of ⎕*FMT*.)

One control pair applied to all columns

If you know that you want all columns formatted in the same way, a single control pair suffices. For example, suppose you have the following data table *Y*, with 3 rows and 3 columns:

```
   Y
21        12345.6789        1.154444
17        12345.6781        3.6789
 6        12345             2.665
```

To have that array *Y* formatted so that every field has 15 print positions and 3 decimals, you might request:

```
      15 3 ⍕ Y
         21.000       12345.679          1.154
         17.000       12345.678          3.679
          6.000       12345.000          2.665
```

Or with each column represented in a field having 15 print positions and no decimals:

```
      15 0 ⍕ Y
             21           12346              1
             17           12346              4
              6           12345              3
```

Field width 0 lets the system pick the width

A special meaning is understood if you state a field width of 0. With ⍕, this does **not** mean that you're asking the system to represent your column in a field 0 characters wide. Rather, it lets the system determine just how many spaces are needed to represent the data, while still leaving at least 1 leading blank. This (along with monadic thorn) is the only use of either format function that yields a result whose shape is not completely determined by the left argument.

Suppose you want to display the same array *Y* shown in the preceding section, but this time with 0 decimals for the first column, 3 decimals for the second columm, and 1 decimal for the last column, but letting the system pick the field width used to represent each column. You could do that by the following instruction:

```
      0 0  0 3  0 1 ⍕ Y
21 12345.679 1.2
17 12345.678 3.7
 6 12345.000 2.7
```

When you let the system pick the field width, but supply a left argument containing only a single pair (for example, `0 2`), the system picks a single width that applies to all the fields; see "A Single System-picked Width for All Columns."

Space for the negative sign

When a value is negative, space is required to write the negative sign. For example, if the element of *Y* in the first row and third column is negative, that column requires an additional print position:

```
      Y[1;3] ← -Y[1;3]

      0 0   0 3  0 1 ⍕ Y
 21 12345.679 ¯1.2
 17 12345.678  3.7
  6 12345.000  2.7
```

However, if the 6 down at the bottom left corner were changed to ¯6, there would be no change in the spacing, since there is already room in that field for its negative sign:

```
      Y[3;1] ← -Y[3;1]

      0 0   0 3  0 1 ⍕ Y
 21 12345.679 ¯1.2
 17 12345.678  3.7
¯6 12345.000  2.7
```

A single system-picked width for all columns

If you use a control pair whose first element is zero (or if you use a scalar), the system selects an appropriate width for that column or columns.

If that pair applies to all columns, then the width that the system picks is a **single** width applied to **all** the columns governed by that pair. The selected width is large enough to accommodate all the numbers in all the columns to which the control pair applies. To be precise, that width is 1 greater than the number of characters required to print the number having the widest representation. The extra 1 assures you of at least one blank between adjacent fields. For example, that same array *Y* can be represented in fixed-point decimal form, with 2 decimals in each field, by an instruction such as this:

```
    0 2 ⍕ Y                    (or alternatively 2 ⍕ Y)
21.00 12345.68   ¯1.15
17.00 12345.68    3.68
¯6.00 12345.00    2.67
```

Leading and trailing zeros with thorn

Leading zeros (to the left of the decimal point) are never displayed, except in the units position. For example:

```
      ⍕ 1÷2
0.5

      8 2 ⍕ 1÷2
  0.50
```

(**Note:** the IBM publication **APL Langauge,** versions 0-3, shows no leading zero in the units position of a field produced by dyadic thorn. When formatted by a system following that rule, a column of fractions each of which has a value less than 1 would require 1 fewer print positions that when formatted by SHARP APL.)

Monadic thorn treats trailing zeros in the same way as unformatted system output. That is, neither leading zeros (to the left of the decimal point) nor trailing zeros (after the decimal point) are supplied.

Dyadic thorn (in either exponential or fixed-point form) shows trailing zeros for the full width available. Thus the number "one half" formatted with dyadic ⍕ to have four decimal places, appears as `0.5000`.

Picking an appropriate field width for decimal displays

To help this discussion of the way a particular field is formatted, call the field width **w** and the number of decimals **d**, so that to format *X* you'd write

```
(w,d) ⍕ X
```

To illustrate, suppose you have a 4-by-3 matrix *X* whose values were established as follows:

```
X ← 4 3 ⍴ * 2 × ¯6+⍳12
```

The display of a column of numbers appears right justified in its field. The last **d** columns contain the digits to the right of the decimal place. If **d** is zero, then there are no decimal places (that is, the numbers are displayed to the nearest integer) and no decimal point.

```
12 0 ⍕ X
```

```
←---w------→←---w------→←---w------→
          0           0           0
          0           0           1
          7          55         403
       2981       22026      162755
```

If **d** is greater than zero, the last **d** columns contain the decimal digits, and the next column leftward contains a decimal point on every line.

```
12 4 ⍕ X
```

```
←----w-----→←----w-----→←----w-----→
      ←d-→        ←d-→        ←d-→
    0.0000      0.0003      0.0025
    0.0183      0.1353      1.0000
    7.3891     54.5982    403.4288
 2980.9580  22026.4658 162754.7914
```

Since both the decimal places and the point are coming out of the total field width **w**, **w must** be greater than **d** (except, of course, for the special case in which **w** is zero).

In the part of the field to the left of the decimal point, you need enough space to write the integer portion of the number having the largest magnitude in that column. You also need an additional column in which to put the negative sign for a value that's negative.

```
      12 4 ⍕ -X
 0.0000     ¯0.0003     ¯0.0025
¯0.0183     ¯0.1353     ¯1.0000
¯7.3891    ¯54.5982   ¯403.4288
¯2980.9580 ¯22026.4658¯162754.7914
```

Unless you really want the adjacent fields printed right up against each other without separation (as they appear in the last row of the preceding example) you should make **w** at least 1 greater than the total space needed for your widest number. The unused space in the field appears as blanks at the left of each field (including at the left of the very first field).

If you want the system to figure out the field width for each column independently, you need a left argument containing a pair of numbers for each column on the right, chosen so that the first member of each pair is zero. That allows a wide field for a column that contains a number whose representation is wide, and a narrow field for one that doesn't.

```
      0 4 ⍕ ⍉X                          (common width for all columns)
      0.0000      0.0183      7.3891   2980.9580
      0.0003      0.1353     54.5982  22026.4658
      0.0025      1.0000    403.4288 162754.7914

      0 4  0 4  0 4  0 4  ⍕ ⍉X          (width set separately for each column)
0.0000 0.0183  7.3891  2980.9580
0.0003 0.1353 54.5982 22026.4658
0.0025 1.0000 403.4288 162754.7914
```

Representing numbers in exponential form

Exponential form is widely used in scientific work to represent very large or very small numbers; it is sometimes referred to as "scientific notation." To write a number in exponential form, you shift the decimal point until the number (whatever its original size) can be written with just **one** digit to the left of the decimal point. Then you show the power of 10 by which that number would have to be multiplied to give you the correct value. Here is an exponential representation of the number 7.3 million:

```
7.3E6
```

The following is 73 million:

```
7.3E7
```

and this is Avogadro's number:

```
6.0228E23
```

The scaled-down portion of the representation, having a value of at least 1 and less than 10, is called the **mantissa.** The power of ten by which the mantissa must be multiplied is called the **exponent** (or sometimes the characteristic). Immediately to the right of the mantissa, a letter *E* (for exponent) appears, followed by the digits that represent the exponent.

A very small number takes a negative exponent. For example, the mass of an electron (in grams) can be written in exponential form as

```
9.1066E¯28
```

The sign of the exponent is independent of the sign of the number itself. As usual, a negative number is indicated by the presence of a negative sign as the very first non-blank character. Thus a very small negative number may have two negative signs in its exponential representation: one for itself (at the beginning) and one for its exponent (right after the *E*). For example the charge of an electron, when displayed, appears as

```
¯1.5921E¯20
```

There's one number that can't be written so that its mantissa is between 1 and 10. That number is zero. By convention, in exponential form, zero is written

```
0E0
```
(or, if you have two decimals) `0.00E0`

Exponential format with dyadic thorn

When the **second** element of a pair in the left argument of ⍕ is **negative,** you get the corresponding column of the right argument represented in exponential form. The absolute value of **d** (that is, the second member of each pair in the left argument) specifies not the number of decimal places but the number of **significant digits.** "Significant digits" includes both the one digit appearing to the left of the point **and** those that come after. For example, if *M* is still the mass of an electron, then formatting it in a field 10 print positions wide, with 4 significant digits, is done like this:

```
      10 ¯4 ⍕ M
 9.107E¯28
```

Or with the same field width and 5 significant digits:

```
      10 ¯5 ⍕ M
9.1066E¯28
```

In each field of the result, both the decimal point and the letter *E* are aligned. In each field, the letter *E* appears in the fourth print position from the right edge of the field, and the last three positions are reserved for the exponent and its sign. Since the exponent may require 1, 2, or 3 positions, the rightmost printed characters in the field may not be in alignment. For example, suppose *X* is a 6-row matrix in which each value is the number three-and-a-third, both positive and negative, and multiplied by various powers of ten:

```
X←6 1⍴ 3.333333333333 × 1 ¯1 ∘.× 10*0 10 ¯10
```

```
      8 ¯2 ⍕ X
 3.3E0
 3.3E10
 3.3E¯10
¯3.3E0
¯3.3E10
¯3.3E¯10
```

System-picked width in exponential form

You can let the system pick the width for an exponential representation, just as you can for a decimal representation:

```
      X←⍉2 3⍴X

      0 ¯2  0 ¯2 ⍕ X
 3.3E0   ¯3.3E0
 3.3E10  ¯3.3E10
 3.3E¯10 ¯3.3E¯10
```

Or you could let the system pick a single width for all fields:

```
      0 ¯2 ⍕ X
 3.3E0   ¯3.3E0
 3.3E10  ¯3.3E10
 3.3E¯10 ¯3.3E¯10
```

Minimum field width for exponential form

To be sure that the result can be written in the space allowed, the field width must be at least 6 greater than the number of digits requested. (That's one position for the point, one for the *E*, two for digits in the exponent, and one each for the sign of the exponent and the sign of the mantissa.) To be sure of having a blank at the beginning of each field, you need a width 7 greater than the number of digits. Provided you know that none of your numbers will be negative, the absolute minimum you can get away with is a width 5 greater than the number of significant digits.

Decimal and exponential fields in the same display

The second member of one pair may be positive while in another pair the second member is negative, so that one column appears in decimal form and another in exponential. Here again is the array *Y* displayed (without explicit formatting) like this:

```
    Y
 21        12345.6789      ¯1.154444
 17        12345.6781       3.6789
¯6         12345            2.665
```

Suppose you need to display *Y* according to the following scheme. The width of each field is to be 14. The first column is to be represented in exponential form with 3 significant digits. The second column is to be represented in exponential form with 8 digits. The last column is to be represented in fixed-point form with 5 decimals:

```
      14 ¯3  14 ¯8  14 5 ⍕ Y
 2.10E1   1.2345679E4        ¯1.15444
 1.70E1   1.2345678E4         3.67890
¯6.00E0   1.2345000E4         2.66500
```

Insufficient space for the result in exponential form

If you request an exponential form to be written in a field too narrow to contain it, the representation of that number is replaced by stars.

```
      X ← 2 1⍴ 1 ¯1 × 10÷3

      10 ¯4 ⍕ X
 3.333E0
¯3.333E0

      9 ¯4 ⍕ X
3.333E0
*********
```

The second row can't be represented because there isn't room for the negative sign. Recall that the last four positions of an exponential field are reserved for the exponent, and aren't avaliable for the mantissa or its sign, even though (as in the example just shown) the exponent only uses two of its four positions.

(**Note:** The convention of using stars to represent a value whose representation won't fit in the width provided is the same in both ⍕ and ⎕*FMT*. By contrast, the IBM publication **APL Language,** version 0-3, treats such a value as a domain error.)

More digits than the system can supply

If you request more digits than the system is capable of providing (for most purposes, 18 is the greatest feasible precision), the additional positions are filled with zeros.

```
      30 ¯23 ⍕ 10÷3
 3.33333333333333330000E0
```

CHAPTER 4

DETAILED DESCRIPTION OF ⎕FMT

The right argument: the data to be formatted

The right argument of ⎕FMT is either an array or a list of arrays. Assume for the moment that the left argument of ⎕FMT can be represented by *F* (for "format"). Then these are examples of possible expressions for formatting a single array:

```
F ⎕FMT X

F ⎕FMT PRICES × INVENTORY - SALES

F ⎕FMT 1976 1977 1978
```

You can also format several arrays together in the same table. When you do that, the various arrays appear in the result beside each other, from left to right across the table.

To format several arrays, make the right argument of ⎕FMT a **list** of the arrays to be formatted. The various members of the list are separated by semicolons, and the entire list must be enclosed in parentheses. For example, to prepare a single table containing part numbers, stock on hand, and unit price, you might have an expression such as the following (still assuming that *F* contains the instructions about how to format each of them):

```
F ⎕FMT (PARTNAME; STOCK; UNITPRICE)
```

If *STOCK* is a table with many columns but you only want their sum in this report, you could write

```
F ⎕FMT (PARTNAME; +/STOCK; UNITPRICE)
```

Because semicolons act as complete separators, you don't have to put parentheses around the individual expressions that are separated by semicolons. On the other hand, you **must** put parentheses around the entire list.

Shape of the arrays to be formatted

Each array to be formatted must be a matrix, or else a vector or scalar. A vector is treated as though it were a one-column matrix, and a scalar as though it were a 1-by-1 matrix. For example, if *X* is simply the numbers 1 to 5, generated by

```
X←⍳5
```

then, while formatting *X* with ⍪ produces a horizontal display:

```
      3 0 ⍪ X
1  2  3  4  5
```

formatting the same array *X* with ⎕FMT produces a vertical display:

```
      'I 3' ⎕FMT X
1
2
3
4
5
```

Watch out. Treating a vector of numbers as a column may not surprise you. In fact, in many situations it's a convenience. However, you may not expect the same rule to apply to a vector of characters. But it does. If you have a character vector (for example, a caption) that you want to display horizontally, this isn't the way to do it. You should form it into a 1-row matrix before you format it, or, if it is constant, include it in the left argument of ⎕*FMT*. Ways of doing this are explained in later sections.

Formatting several arrays when some require more rows than others

The formatting function ⎕*FMT* doesn't object if the arrays to be formatted have different numbers of rows. The final result is a matrix with as many rows as there are in the array with the most rows.

Suppose that, when formatted, array *A* appears as a 4-by-12 table each element of which is the letter *A*, and that *B* is 6-by-8 table containing *B*s, and *C* is a 2-by-20 table of *C*s. Then the result of formatting all three of them is a matrix having 12+8+20 print positions (that is, 40 print positions), and 6 rows (since that's how many rows there are in *B*, the one with the greatest number of rows):

```
      F ⎕FMT (A; B; C)
AAAAAAAAAAAABBBBBBBBCCCCCCCCCCCCCCCCCCCC
AAAAAAAAAAAABBBBBBBBCCCCCCCCCCCCCCCCCCCC
AAAAAAAAAAAABBBBBBBB
AAAAAAAAAAAABBBBBBBB
            BBBBBBBB
            BBBBBBBB
```

The result is a rectangular array, 6-by-40, in which the unused rows below the short fields are filled with blanks.

The left argument: the format control vector

The function ⎕*FMT* always requires a left argument. Its left argument must be a character vector.

The left argument consists of one or more formatting **phrases.** Generally speaking, each phrase describes a **field** of the result, by specifying how to format a **column** from the right argument. (The exception is that certain phrases insert text, or control the positioning, but don't themselves process a column from the right argument.)

The columns of the right argument are treated one by one, in order from left to right. If several arrays are being formatted together, for example, by an expression such as

```
      F ⎕FMT (A; B; C)
```

then ⎕*FMT* processes each of the columns of *A*, and then each of the columns of *B*, and finally each of the columns of *C*. The various phrases within the left argument are applied in turn, straight across the entire collection of columns, regardless of which array they came from, the first phrase to the first column, the second to the second, and so on.

Formatting phrases are reused cyclically. If you provide **fewer** phrases than there are columns, after the last phrase has been used, processing of the next column starts over with the first phrase, and so on. Similarly, if you provide **more** formatting phrases than there are columns, the excess phrases are ignored.

Separating one phrase from the next

Within the left argument of ⎕*FMT*, the various phrases are separated by commas. If you ignore what's in the individual phrase and for the moment represent them by xxxx, a left argument containing four format phrases might look like this:

```
'xxxxx,xxxxxxxxxxxxxxxx,xxxx,xxxxx'
```

Except inside a literal text, blanks in the left argument of ⎕*FMT* have no effect. So it's all right to put spaces around the commas if that helps the readability of the sequence of phrases:

```
'xxxxx, xxxxxxxxxxxxxxxx , xxxx , xxxxx '
```

In ⎕*FMT*'s interpretation of its left argument, the commas act as complete separators. The phrases do not interact with each other. Each is interpreted on its own.

Repeated phrases

If the same phrase will serve for **all** the columns of the right argument, you only need to state it once. It is applied repeatedly for all the columns. Similarly, if the same three phrases can be used over and over, you need only supply those three phrases. With ⎕*FMT*, there is no requirement that the number of phrases divide evenly into the number of columns of data.

More complicated repetitions can be shown by inserting an explicit **repetition factor.** If the first non-blank character (or characters) in a phrase are numeric, then the phrase is to be applied that many times. For example, a left argument might be constructed like this:

```
'3 xxxxx, xxxxxx, 2xxxxxxx'
```

That would mean that the first phrase is to apply to each of 3 columns, the next phrase to 1 column (since no number was shown), and the last to 2 columns. This cycle is therefore good for 6 columns, and can then repeat until all columns have been formatted.

If a sequence of several phrases is itself repeated, you can show that by enclosing them in parentheses, and putting the repetition factor to the left of the opening parenthesis. For example, here's a sequence of two phrases which is repeated 3 times, followed by two phrases that are not repeated:

```
'3(xxxxx,xxxxx),xxxx,xxxxxx'
```

The threefold repetition of the first two phrases takes care of the first six columns of data. The last two phrases therefore apply to columns 7 and 8.

Watch out. Format phrases may contain text material that is set off by special delimiting characters. ⎕*FMT* recognizes several such delimiters (see the next section). Anything enclosed by these delimiters is counted as part of a text to be printed, and does **not** serve as a delimiter. Commas, parentheses and repetition numbers have the functions described here **only when not enclosed by text delimiters.**

Watch out. When parentheses enclose a phrase which consists solely of literal text, they have a special significance in deciding whether the text should appear in the formatted result. See the section entitled "Inserting a constant text."

Literal text within a format phrase

In many situations, the format phrase contains a fragment of text that is to appear in the formatted display. To set that text off from the rest of the phrase, it is surrounded by text delimiters. They don't themselves appear in the finished display, but serve in the format phrase to mark the boundaries of the phrase.

Several different characters are recognized as text delimiters. The choice is provided so you can pick symbols that don't conflict with the text that you want printed. Note that the single quote mark ' that is used in other APL expressions is **not** one of the text delimiters recognized by ⎕*FMT*.

The ⎕*FMT* text delimiters are:

<	>	(less than, greater than)
⊂	⊃	(subset, superset)
¨	¨	(dieresis)
⎕	⎕	(quad)
⍞	⍞	(quad-prime)

Notice that the first two are used in matched pairs, with an opening and closing symbol, whereas the others use the same symbol to mark both the start of a text and its end.

Reading the format control vector from left to right, a literal text runs from the first use of an opening text delimiter until the **next** use of the corresponding end symbol. Text delimiters can't be nested.

If a literal text starts with <, the literal text continues until the next use of >. If it starts with ⊂, the literal text runs until the next use of ⊃. If it starts with ¨, the literal text runs until the next If it starts with ⎕, the literal text runs until the next ⎕. If it starts with ⍞, the literal text runs until the next ⍞.

Within the zone thus enclosed, **any** character is taken as part of the text. That means that if you want to print one of the characters that is used as a delimiting symbol, you can do so by putting it inside a segment enclosed by a pair of the other text delimiters. It also means that if you inadvertently place text delimiters so that they surround the commas or key letters needed as part of your format control, what you intended as control information has no effect, but is instead treated as part of the text.

The structure of a format phrase

At the heart of each format phrase is a single letter that specifies what sort of a phrase it is. (The only exception is the text insertion phrase, which consists of nothing but a string of text entirely surrounded by text delimiters.) Within a phrase, the one letter that identifies its type is called the **key letter.**

You might think of that key letter as a function. It always has a right argument (that is, some information further to the right of it). There may also be a left argument, consisting of one or more sub-phrases, to the left of the key letter.

The key letter does not need to be set off from the rest of the phrase in any way. It doesn't need spaces around it. On the other hand, except as part of a literal text, spaces in the left argument of ⎕*FMT* are not significant, so putting spaces around the key letter won't hurt, and may make the phrase easier to read. Most of the examples in this manual show a space on either side of the key letter.

The "right argument" of a format phrase

To the right of the key letter you write information that fixes the width of the field. This appears as a number in every type of phrase but one. The exception is the *G* phrase, which supplies a "picture" of the way the field is to look, enclosed in text delimiters. In that case, the width of the field is the width of the text. In the *E* and *F* formats, a second number to the right of the key letter determines the number of decimal places.

To illustrate, following this paragraph, a possible format vector is shown. It contains eight phrases, one of each of the eight types. In these illustrative phrases, there are only a few instances of qualifiers or decorators. The numbers below the sample vector are positioned under the key letter in each phrase. A brief explanation of each appears in the list of eight items below them.

```
'G<999(9)>, X4, CBI12, T34, R<*>M<- >F10.2, 10A1, <TEXT>, E10.6'
 ↑          ↑    ↑    ↑              ↑        ↑   ↑       ↑
 1          2    3    4              5        6   7       8
```

1. Picture. Replace the 9s with the appropriate digits.

2. Move 4 positions to the right before starting the next field.

3. Display as integers in a field 12 positions wide.

4. Move to position 34 to start the next field.

5. Display as fixed-point, in a field 10 positions wide, with 2 decimals.

6. Display 10 columns of characters, single spaced.

7. Display the text between the delimiters.

8. Display in exponential form, in a field 10 positions wide with 6 significant digits.

When you are reading a format phrase, your first task is to spot what sort of a phrase it is. The key letter is the last thing in the phrase towards its right end, followed only by a number (phrases *X T A E I F*) or by a string of text enclosed in text delimiters (phrase *G*).

Types of "left argument" to a format phrase

To the left of the key letter in a format phrase, there may appear two types of things: **qualifiers** and **decorators.** Just which may appear with which phrases is summarized in Table 3. They can appear in any order (as long as they're all to the left of the key letter), and they don't need any special separation from each other (although in this manual the examples usually show them set off from their neighbours by spaces).

The effect of each qualifier or decorator is described in detail in the sections that follow. They are listed below:

List of ⎕*FMT* Qualifiers and Decorators

Qualifiers:

*K*nn	Scale factor. Multiplies values by 10*nn before formatting.
B	Blanks entries whose value is zero.
Z	Supplies leading zeros.
C	Inserts commas between groups of three digits.
L	Left justifies numbers.

Decorators:

R <*TEXT*>	Background text to fill unused positions otherwise available to represent the number.
M <*TEXT*>	Leading text, to left of a negative number.
N <*TEXT*>	Trailing text, to right of a negative number.
P <*TEXT*>	Leading text, to left of a non-negative number.
Q <*TEXT*>	Trailing text, to right of a non-negative number.
S <$\alpha\omega$>	Substitute character ω for standard character α.

Qualifiers and Decorators
and the Formats With Which They Are Available

		G	*I*	*F*	*E*	*A*	*<TEXT>*	*X*	*T*
Scale	*K*	Yes	Yes	Yes	Yes	---	---	---	---
Blank 0-value	*B*	Yes	Yes	Yes	Yes	---	---	---	---
Leading zeros	*Z*	No	Yes	Yes	Yes	---	---	---	---
Commas	*C*	No	Yes	Yes	Yes	---	---	---	---
Left justify	*L*	No	Yes	Yes	Yes	---	---	---	---
Background	*R*	Yes	Yes	Yes	Yes	Yes	---	---	---
Left, <0	*M*	Yes	Yes	Yes	Yes	---	---	---	---
Right, <0	*N*	Yes	Yes	Yes	Yes	---	---	---	---
Left, ≥0	*P*	Yes	Yes	Yes	Yes	---	---	---	---
Right, ≥0	*Q*	Yes	Yes	Yes	Yes	---	---	---	---
Substitute	*S*	Yes	Yes	Yes	Yes	---	---	---	---

Note: A combination marked "---" is not prohibited, but has no effect.

Format by picture

With the *G* format phrase, you provide a model text (or picture) which illustrates the way you want the field to look. Within that picture, you use the digit 9 or the letter Z to mark the positions at which you want to have digits inserted. The system replaces the 9s and Zs with the digits that represent the number. The rest of the picture remains unchanged—except as will be noted in a moment. The width of the resulting field is exactly the width of the picture.

The positions at which digits are printed don't have to be consecutive. The *G* format is therefore particularly useful when you want to space out the representation of a number in some special way, or insert arbitrary characters between the digits. For example, Canadian and US telephone numbers consist of ten digits, of which the first three are an area code (usually shown in parentheses), and the next three are an exchange, usually separated from the last four by a hyphen. You could get that format for some numbers called *NOS* by the following:

```
      NOS←4163645361 4163632051

      'G <(999) 999-9999>' ⎕FMT NOS
(416) 364-5361
(416) 363-2051
```

Notice that the entire left argument of ⎕*FMT* is enclosed in APL quotes (that is, single quotes), and that within it, the text of the picture is enclosed between the text delimiters < and >.

The same pair of numbers could be inserted into a more elaborate text:

```
      'G <IN AREA 999, PLEASE DIAL 999-9999>' ⎕FMT NOS
IN AREA 416, PLEASE DIAL 364-5361
IN AREA 416, PLEASE DIAL 353-2051
```

Decimal point in picture format

The digits that the system inserts into a *G* format phrase are the decimal digits needed to represent the value in the right argument **to the nearest integer.** If you include a decimal point in the picture, it appears in the result. But to the system, the decimal point in the picture is just a text character. It has no significance in scaling the number, or in aligning the digits.

To format fractions, do the following:

1. Write an appropriate number of 9s or Zs to the right of the decimal point in the picture.

2. Use the scaling qualifier *K* so that the value, when rounded to the nearest integer, fills the digits appropriately.

Suppose you have amounts of money stored as dollars, including amounts such as 1.17 to represent "one dollar and 17 cents." You want to show them with two decimal places. Use a picture showing two decimal digits to the right of the decimal point **and** use the scale factor *K2* to shift the decimal point in the argument by two places before formatting:

```
      AMT←100 1.17 77.88 .6666666

      'K2 G<ZZ9.99>' ⎕FMT AMT
100.00
  1.17
 77.88
  0.67
```

In many commercial applications, amounts of money are stored internally as integers, and represent cents rather than dollars. In that case, to show them as dollars and cents in the report, you still need two places to the right of the decimal in the picture, but no scale factor:

```
      CENTS←10000  117  7788  67

      'G <ZZ9.99>' ⎕FMT CENTS
100.00
  1.17
 77.88
  0.67
```

If you deal with large numbers, you might want to show them to the nearest million dollars:

```
      BIG←1234560000  1000000  70000  227000000
```

To shift the decimal point 6 places to the left (in effect, to divide the amounts by a million before formatting) use the scale factor ¯6, which may be written either as *K*¯6 or as *K*-6:

```
      'K-6 G<ZZZ,ZZ9 MILLION>' ⎕FMT BIG
1,235 MILLION
    1 MILLION
    0 MILLION
  227 MILLION
```

If you want those same amounts represented to the nearest tenth of a million, put one digit to the right of a decimal point in the picture, and use a scale factor of ¯5 instead of ¯6:

```
      'K-5 G<ZZZ,ZZ9.9 MILLION>' ⎕FMT BIG
1,234.6 MILLION
    1.0 MILLION
    0.1 MILLION
  227.0 MILLION
```

Leading and trailing zeros in picture format

At each position where your picture shows a 9, the system inserts one of the digits 0-9. A 9 in the picture is never replaced by a blank. Nor can a 9 position be used for the sign, or for any other optional decorator you may have specified.

A position marked by a Z may also be used to write digits representing the number. It differs from a position marked by a 9 in the following way. If the digit to be printed is a leading or a trailing zero, then the position is free to be used for the sign or for a decorator. If it's not needed for either of those, the position is left blank.

A position marked by a Z is free (that is, available for a blank or a decorator) provided there is no printing digit between it and the edge of the field. Since a position marked by a 9 **always** results in a printed digit, a Z that has 9s on both sides of it can't have any effect different from the effect of a 9. Putting it the other way, it only makes sense to use Zs at the edges of the pattern of 9s and Zs. Leading Zs give you leading zero suppression, and trailing Zs give you trailing zero suppression. The examples in the preceding paragraphs showed suppression of leading zeros.

Text characters in the picture that have Zs on both sides of them are reproduced only when those positions are used in the result for some character other than a blank. Stating it the other way, an interpolated text character in the picture is not printed if the digit positions surrounding it are both blank. That was evident in the examples used in the preceding section, where a comma was inserted between Zs.

Position of negative sign or decorators in picture format

The negative sign overwrites the first available position to the left of the leftmost digit.

A position is available for the sign either because it was marked by a Z in the picture, or because it was a part of the text provided in the picture. In the examples that follow, the message →→→→→→→→→ and ←←←←←←←←← are used to show arbitrary text on either side of the zone used for numbers. Notice where the negative sign is placed:

```
NOS←¯123456 ¯12345 ¯1234 ¯123 ¯12 ¯1
```

```
      'G <→→→→→→→→→ZZZ,ZZ9←←←←←←←←←←>' ⎕FMT NOS
→→→→→→→→ ¯123,456←←←←←←←←←
→→→→→→→→→ ¯12,345←←←←←←←←←
→→→→→→→→→  ¯1,234←←←←←←←←←
→→→→→→→→→    ¯123←←←←←←←←←
→→→→→→→→→     ¯12←←←←←←←←←
→→→→→→→→→      ¯1←←←←←←←←←
```

The negative sign can overwrite the → that's next to the leftmost *Z*, or text embedded within the zone used for digits (such as the comma in this example). But all the digits must be able to fit into the positions marked by `9` or *Z*. Digits are never allowed to overwrite any of the other positions.

Sign-dependent decorators with picture format

Unless you indicate otherwise, the system marks a negative number by placing a negative sign ¯ to the left of its leftmost digit. It leaves non-negative numbers unmarked. There are four optional decorators that you can use instead. Each consists of a single letter followed by text enclosed in text delimiters. If these are used, they may appear in the format phrase in any order, provided they're to the **left** of the key letter. The four sign-decorator codes are:

P <TEXT> Leading text, to the left of the leftmost printing digit of a non-negative number.

Q <TEXT> Trailing text, to the right of the rightmost printing digit of a non-negative number.

M <TEXT> Leading text, to the left of the leftmost printing digit of a negative number (replaces the negative sign).

N <TEXT> Trailing text, to the right of the rightmost printing digit of a negative number.

In *G* format, these decorators overwrite the text provided in the picture, including positions marked *Z* and not needed to print a digit, but **not** positions marked with a `9`.

```
      NOS←123456 ¯123456 123 ¯123 0

      'M<MINUS > Q< PLUS> G<→→→→→→→→→ZZZ,ZZ9←←←←←←←←←←>' ⎕FMT NOS
→→→→→→→→→123,456 PLUS←←←←←
→→→MINUS 123,456←←←←←←←←←
→→→→→→→→→    123 PLUS←←←←←
→→→→→→→MINUS 123←←←←←←←←←
→→→→→→→→→      0 PLUS←←←←←
```

Floating dollar sign in picture format

The sign-dependent decorators "float." That is, they begin at the next print position not used for a digit, rather than having a fixed vertical alignment. Hence the way to print a floating dollar sign is to include it as part of each of the decorators that can appear on the left (that is, *P* and *M*). For example:

```
      'P< $> M< $> Q<+ > N<- > G<→→→→→→→→→ZZZ,ZZ9←←←←←←←←←>' ⎕FMT NOS
→→→→→→→ $123,456+ ←←←←←←←←
→→→→→→→ $123,456- ←←←←←←←←
→→→→→→→→   $123+ ←←←←←←←←
→→→→→→→→   $123- ←←←←←←←←
→→→→→→→→     $0+ ←←←←←←←←
```

Trailing zeros and blanked-out field for zero value

If the digit positions are marked entirely by Zs, then neither leading nor trailing zeros are printed. (This is a rather unusual situation: it means that the trailing zeros in numbers such as 10 or 1000 don't appear.) In that case, a value of zero is represented solely by blanks. If **no** digits at all are used to represent the value 0, the decorators to which it might be entitled (as a non-negative number) don't appear either:

```
      NOS←10000 ¯10000 1000 ¯1000 100 ¯100 10 ¯10 1 ¯1 0

      'M< $> P< $> Q<+ > N<- > G<→→→→→→→→→ZZZ,ZZZ←←←←←←←←←>' ⎕FMT NOS
→→→→→→→→ $1+     ←←←←←←←←←
→→→→→→→→ $1-     ←←←←←←←←←
→→→→→→→→→ $1+    ←←←←←←←←←
→→→→→→→→→ $1-    ←←←←←←←←←
→→→→→→→→→   $1+ ←←←←←←←←←
→→→→→→→→→   $1- ←←←←←←←←←
→→→→→→→→→    $1+ ←←←←←←←←
→→→→→→→→→    $1- ←←←←←←←←
→→→→→→→→→     $1+ ←←←←←←←
→→→→→→→→→     $1- ←←←←←←←
→→→→→→→→→       ←←←←←←←←←
```

Field overflow with picture format

If the digits of the number require more space than has been provided, a field overflow occurs. You also get a field overflow if there isn't enough room outside the positions used for digits in which to insert the negative sign or the optional decorators. However, overflow occurs only if the negative sign or decorators are in fact needed to represent the numeric values in the right argument; setting up a picture which **would** fail under some conditions doesn't prevent ⎕*FMT* from handling other data in which the problem doesn't arise.

If there is a field overflow, the entire field is replaced by stars, including not only the positions marked by 9 or Z, but also the rest of the picture as well. For example:

```
      NOS←¯1234567 ¯123456 ¯12345 123 12345

      'P<POSITIVE> G <→→→→ZZZ,ZZ9←←←←>' ⎕FMT NOS
***************
→→→¯123,456←←←←
→→→→¯12,345←←←←
POSITIVE123←←←←
***************
```

Field overflow occurred for the first number because, although there was space for its negative sign, there wasn't room for its seven numeric digits. Field overflow occurred for the last number because there wasn't room to print the decorator *POSITIVE* to the left of the leftmost printing digit.

Background text to fill Z positions left blank

The decoration code *R* lets you provide a background text that serves to fill any of the positions marked by a Z in the picture, but which (in your picture) are needed neither for digits nor for any part of the sign or the sign-dependent decorators.

Think of the background text as running across the **entire** field, but showing through only where there are Zs (or text surrounded by Zs) that don't get filled by anything else. Blanks that are explicitly included within a decorator **do** obliterate the background text.

To make clear which positions from the background text show through, consider the (otherwise rather uninteresting) background text consisting of the 26 letters of the alphabet *ABCDEFGHIJKLMNOPQRSTUVWXYZ*. Then suppose you elect to mark each positive value with the decorator < + > ; that is, a plus sign with a blank on either side of it. Here is a format vector *F*:

```
      F
R<ABCDEFGHIJKLMNOPQRSTUVWXYZ> P< + > G <→→→→→ZZZ,ZZZ,ZZZ,ZZ9←←←←←>
```

Apply that to the numeric vector *NOS*, whose values have been deliberately chosen to cover a wide range, so as to illustrate where background characters appear and where they are obscured:

```
      NOS←1.23456789E10  1234567890  123456779  123  12  1

      F ⎕FMT NOS
→→→ + 12,345,678,900←←←←
→→→→ + 1,234,567,890←←←←
→→→→→F + 123,456,779←←←←
→→→→→FGHIJKLMN + 123←←←←
→→→→→FGHIJKLMNO + 12←←←←
→→→→→FGHIJKLMNOP + 1←←←←
```

Notice that the letters *ABCDE* **never** appear, because they are always obscured either by the arrows contained in the main picture, or by the sequence < + > contained in the decorator. Similarly, the letters *UVWXY* never appear either, since they are always obscured by the ← arrows from the main picture. And the 26th letter of the background text never gets a chance to appear because there are only 25 positions in the entire field.

The background text shows through **only** where there are Z positions left blank, or commas surrounded by Z positions left blank. If the final 9 in the picture is changed to a Z, the background shows through in the trailing positions vacated in the first and second rows:

```
      F
R<ABCDEFGHIJKLMNOPQRSTUVWXYZ> P< + > G <→→→→→ZZZ,ZZZ,ZZZ,ZZZ←←←←←>
```

```
      F ⎕FMT NOS
→→→ + 12,345,678,9ST←←←←←
→→→→ + 1,234,567,89T←←←←←
→→→→→F + 123,456,779←←←←←
→→→→→FGHIJKLMN + 123←←←←←
→→→→→FGHIJKLMNO + 12←←←←←
→→→→→FGHIJKLMNOP + 1←←←←←
```

Substituting other characters for 9 or Z in the picture, or for 0 or * in the result

When combined with *G* format, the *S* decorator permits you to substitute characters of your own choice for those used with a fixed meaning either in the picture controlling that field, or in the display of that field. In *G* format, only two characters have fixed meanings in the picture: 9, the digit selector, and Z, the digit selector with suppression of leading or trailing zeros. In the result, the following characters have fixed meanings: * is used for field overflow, and zero is used to fill insignificant leading positions marked by 9s in the picture.

You can declare your own substitutes for those by using the *S* decorator code. The letter *S* must appear within the format phrase to the left of the key letter, and followed by a text string enclosed in text delimiters. That string must contain an even number of characters. In each pair, the first member must be one of the characters 9Z*0. The second member may be whatever you like, except that you can't replace both 9 and Z by the same character. Consider an example. To begin with, suppose the picture itself contains text such as 1977 *ZONE TOTAL*. In that case, neither the 9 nor the Z is intended to serve as a digit selector. You need some other characters to mark the digit positions. Let those be ○ for 9 and ∘ for Z. Then suppose that for field overflow the symbol ◇ is preferred, and that leading zeros are to be filled not with a zero but with a colon. The format control vector is then:

```
      F←'S<9○Z∘*◇0:>  G <1977 ZONE TOTAL→→→→→○○○,○○○.∘∘∘←←←←←>'
      NOS←1234567890  1234567  123456  123000

      F ⎕FMT NOS
◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇
1977 ZONE TOTAL→→→→→:::,123.456←←←←←
1977 ZONE TOTAL→→→→→:::,123    ←←←←←
```

You can see in the example above that the leading zeros (whose positions were marked by ○s acting as substitutes for 9s) now appear as :, but the trailing zeros (marked by ∘ acting as substitutes for Zs) still yield blanks. You could use a blank as an explicit replacement for the leading zero. That is illustrated in the following format vector:

```
      FF←'S<9○Z∘*◇0 >  G <1977 ZONE TOTAL→→→→→○○○,○○○.∘∘∘←←←←←>'

◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇
1977 ZONE TOTAL→→→→→   ,123.456←←←←←
1977 ZONE TOTAL→→→→→   ,123    ←←←←←
```

Notice that under these circumstances, characters embedded within the zone used for digits (in this example, commas) still appear in the third and fourth rows of the result, just as they did in the preceding example when the leading zero was represented by the : character. That's because an embedded character is replaced by a blank only when the position to the left of it is marked in the picture by a Z (or whatever is substituted for Z). Blanks arising as substitutes for a leading zero (in a position marked by a 9 or whatever is substituted for 9) don't affect embedded characters.

The integer format *I* and fixed-point format *F*

These two formats are governed by similar rules, and so they're described together. In particular, the same qualifiers and decorators apply to them.

The integer format *I* is indicated by a phrase containing the key letter *I*, followed by a number. The number is the width of the field in which the numbers are to be represented. If the numbers are not in fact integers, the result shows them represented to the nearest integer.

```
      NOS←(100 10 1 + 2÷3), .5
      NOS
100.6666667  10.66666667  1.666666667  0.5

      'I5' ⎕FMT NOS
  101
   11
    2
    1
```

The *F* format is indicated by a phrase containing the key letter *F*, followed by two numbers, separated by a period. The first number is the width of the field. The second number is the number of decimal places. Thus *'F20.12'* means that the field is to be 20 characters wide, and is to contain 12 digits to the right of the decimal point. A field produced by an *F* format phrase **always** includes a decimal point (except for a row that is completely blank, or a row entirely replaced by stars to indicate field overflow). That is true even if the number of decimal places requested is zero. For example:

```
      'F8.2' ⎕FMT NOS
  100.67
   10.67
    1.67
    0.50

      'F8.0' ⎕FMT NOS
   101.
    11.
     2.
     1.
```

In either format, the representation of the numbers appears right justified in the field (unless you specifically direct otherwise by using the qualifier *L*, discussed later in this chapter).

The field must be wide enough to contain all the needed characters, including the sign and such optional decorators as you may have requested. If there are not enough print positions to contain the representation, you have a **field overflow.** In that case, the corresponding row of that field in the result is filled with stars. For example:

```
      'F8.2' ⎕FMT 1000×NOS
********
10666.67
 1666.67
  500.00
```

Of course, if you specify that a field has 0 width, that's too small to contain any representation, but you won't see the stars— you'll have an empty field of stars.

Exponential form with ⎕*FMT* format *E*

The exponential representation shows every number with one digit to the left of the decimal place, and then, after a letter *E*, the power of ten by which that number must be multiplied to give you the correct value. Some instances of exponential form were shown in the discussion of dyadic thorn. In addition, exponential formatting with ⎕*FMT* may be modified by qualifiers or decorators in the same way as *I* and *F* formats.

You request the exponential form by a formatting phrase in which the key letter is *E*, followed by two numbers separated by a decimal point. The first of those numbers is the total width of the field. The second is the number of significant digits to be displayed. For example:

```
      NOS←6028800000000000000000000 ◇ NOS←NOS, 0, ÷NOS

      'E 10.3' ⎕FMT NOS
  6.03E23
  0.00E0
  1.66E¯24
```

With the *E* format, four positions immediately following the last digit of the mantissa are reserved for the letter *E* (or whatever character you substitute for it) and the exponent. The three exponent positions are filled from the left. They contain the sign of the exponent (represented by ¯ or whatever character you substitute for it) and one or two digits representing the magnitude of the exponent. Since the exponent's magnitude may require only one digit, and may not require a negative sign, one or two of these positions may be unused. If so, they're left blank.

If you ask for two or more significant digits, you also get a decimal point. One of the digits appears to the left of the point, the others to the right. The decimal points are aligned.

```
      'E 8.3' ⎕FMT NOS
6.03E23
0.00E0
1.66E¯24
```

If you ask for only one significant digit, you don't get a decimal point. If you ask for zero significant digits, you get only the letter *E* and the exponent.

```
      'E 8.1' ⎕FMT NOS
  6E23
  0E0
  2E¯24
```

```
      'E 8.0' ⎕FMT NOS
   E23
   E0
   E¯24
```

If the value being represented is negative, a high negative sign appears in the first available position to the left of the characters used to represent the absolute value of the number:

```
      'E 10.3' ⎕FMT -NOS
 ¯6.03E23
  0.00E0
 ¯1.66E¯24
```

Field overflow occurs if the total width prescribed for the field is insufficient to contain the four positions needed for the exponent, the number of significant digits, the decimal point (if you asked for two or more digits) and the sign (if needed by the data). When a field overflow occurs, the entire width of the field is replaced by stars.

```
      X← (1 2 3 ÷ 3) ∘.× 1 0 ¯1
      X
 0.3333333333  0          ¯0.3333333333
 0.6666666667  0          ¯0.6666666667
 1             0          ¯1

      'E 10.4' ⎕FMT X
 3.333E¯1  0.000E0  ¯3.333E¯1
 6.667E¯1  0.000E0  ¯6.667E¯1
 1.000E0   0.000E0  ¯1.000E0

      'E 10.5' ⎕FMT X
3.3333E¯1 0.0000E0  **********
6.6667E¯1 0.0000E0  **********
1.0000E0  0.0000E0  **********

      'E 10.6' ⎕FMT X
**********
**********
**********
```

Since the representations are built from the right edge of each field, if you specify a total width substantially greater than needed for the precision you requested, the unused positions appear as blanks at the left side of the field:

```
      'E20.5' ⎕FMT X
          3.3333E¯1          0.0000E0          ¯3.3333E¯1
          6.6667E¯1          0.0000E0          ¯6.6667E¯1
          1.0000E0           0.0000E0          ¯1.0000E0
```

Alignment of columns

In the preceding examples showing formats *I*, *F*, and *E*, the units position on each row fell in the same column throughout a display. Where there were decimal points, they were aligned one above the other. However, that is **not** automatically the case. It arises because, within each example, the numbers happened to be represented with the same number of decimal places, were right justified in the field, and because **in those particular examples there was nothing further to the right of the rightmost digit.**

If you were to specify sign-dependent decorators on the right (using the *N* or *Q* decorators) and if those decorators differed in length, the alignment of the columns of digits would be lost. In some situations you might seek deliberately to align negative and non-negative values differently. But if you want to preserve alignment, then you need to choose decorators on the right that are of equal length, or else avoid them altogether.

With *E* format, the four positions devoted to the exponent, its sign, and the letter *E* are ordinarily the last four positions of the field. However, if you use the trailing decorators *N* or *Q*, those decorations appear further to the right, following the positions reserved for the exponent. For example:

```
      'Q<POS> N<NEGATIVE> E 18.3' □FMT NOS, -NOS
       6.03E23 POS
       0.00E0  POS
       1.66E¯24POS
¯6.03E23 NEGATIVE
       0.00E0  POS
¯1.66E¯24NEGATIVE
```

The point to remember regarding vertical alignment is this: the controlling factor is that the system right-justifies the **characters** you use, rather than the units position or the decimal point.

Qualifiers usable with *I*, *F*, and *E* formats

The following qualifiers can be used with *I*, *F* or *E* format:

*K*nn Scale factor

L Left justify

B Zero value shown as blank

Z Leading zeros inserted

C Commas inserted between sets of 3 digits to the left of the decimal point.

The letters representing any of these must be to the left of the key letter. Within that zone, they may be in any order, and need not be separated from each other in any way. The examples shown here usually show them separated by spaces, but that's just to improve readability.

Scale Factor *K*

The qualifier *K* in effect shifts the decimal point in the representation. It is as though the column of numbers being represented were multiplied by 10 raised to the power indicated by the numbers that follow the letter *K*. For example, *K2* causes the result to show numbers 100 times larger, while *K¯2* shows them 100 times smaller. For convenience, □*FMT* permits the negative sign in a *K* qualifier to be written either with the high negative sign ¯ or the minus sign -. For example:

```
      NOS
100.6666667  10.66666667  1.666666667

      'K3 I8' □FMT NOS
  100667
   10667
    1667

      'K¯5 F12.8' □FMT NOS
  0.00100667
  0.00010667
  0.00001667

      'K¯5 E15.8' □FMT NOS
  1.0066667E¯3
  1.0666667E¯4
  1.6666667E¯5
```

This use of the *K* qualifier is identical to its use with *G* format, described earlier.

Left justifying with qualifier *L*

The presence of the qualifier *L* causes the characters of the result to be aligned with the left margin instead of the right. The first character position is occupied by the negative sign (if there is one), or by the first character of the leading decorator (if there is one). If there is no sign or decorator, the first position is used by the first significant digit representing the number. Positions left unused at the right side of the field are filled with blanks.

```
      'L I5' □FMT NOS
101
11
2

      'L F12.5' □FMT NOS
100.66667
10.66667
1.66667
```

Since significant characters (either the sign, the decorator, or the first digit of the result) start immediately at the left margin, there are never leading zeros. When you use the *L* qualifier, using the *Z* qualifier at the same time has no additional effect.

Blank for zero value with *B*

The qualifier *B* causes the row used to represent a value of zero to be blank. Not only is the 0 omitted, but so also is the decimal point (in *F* format), leading zeros (if they've been requested—see the next paragraph), and decorators which might otherwise have been printed before or after the representation of the number.

```
      X←123.456  1  0  ¯1  0.123456  654.321

      'B F 12.8' ⎕FMT X
123.45600000
  1.00000000

 ¯1.00000000
  0.12345600
654.32100000

      'B I3' ⎕FMT X
123
  1

 ¯1

654

      'B Z F12.8' ⎕FMT X
123.45600000
001.00000000

¯01.00000000
000.12345600
654.32100000
```

Leading zeros with *Z*

The qualifier *Z* causes zeros to be inserted in unused positions to the left of the leftmost significant digit. Positions available to be filled in this manner can't arise if the field is formatted left justified.

```
      'Z I3' ⎕FMT NOS
101
011
002

      'Z F 12.8' ⎕FMT NOS
100.66666667
010.66666667
001.66666667
```

```
      'Z E15.8' ⎕FMT NOS
001.0066667E2
001.0666667E1
001.6666667E0
```

When the number requires a negative sign, or a leading decorator, the zeros are inserted in any remaining vacant position **between** the sign or decorator and the digits of the number itself. In effect, with the Z qualifier, the negative sign and leading decorators are set left justified, while the digits and trailing decorators are set right justified. The leading zeros fill any remaining spaces in the middle.

```
      'Z F15.8' ⎕FMT X
000123.45600000
000001.00000000
000000.00000000
¯00001.00000000
000000.12345600
000654.32100000

      'Z F15.8' ⎕FMT -X
¯00123.45600000
¯00001.00000000
000000.00000000
000001.00000000
¯00000.12345600
¯00654.32100000

      'Z P<PLUS> M<-> F20.8' ⎕FMT X
PLUS0000123.45600000
PLUS0000001.00000000
PLUS0000000.00000000
-0000000001.00000000
PLUS0000000.12345600
PLUS0000654.32100000

      'Z P<PLUS> M<-> F20.8' ⎕FMT -X
-0000000123.45600000
-0000000001.00000000
PLUS0000000.00000000
PLUS0000001.00000000
-0000000000.12345600
-0000000654.32100000
```

Insertion of commas with *C*

The readability of large numbers can often be improved by inserting a comma between each group of three digits, so that (for example) one million appears not as 1000000 but as 1,000,000. This is provided in *I* and *F* formats by the qualifier *C*.

```
      NOS←123456789.1234567 12345678.12345678 123456.123456
```

```
      'F20.7' ⎕FMT NOS
  123456789.1234567
   12345678.1234568
     123456.1234560

      'C F20.7' ⎕FMT NOS
123,456,789.1234567
 12,345,678.1234568
    123,456.1234560
```

Notice that commas are inserted only to the left of the decimal point. A comma appears only if there's a digit printed on each side of it. (But if you use qualifier *Z* to supply leading zeros, you get commas in **all** the eligible positions, even if one turns out to be the leftmost column in the field.)

A negative sign (or a leading decorator) can appear in a position otherwise used for a comma:

```
      'C F20.7' ⎕FMT -NOS
¯123,456,789.1234567
 ¯12,345,678.1234568
    ¯123,456.1234560
```

The character inserted by the *C* qualifier is not necessarily a comma. Using the substitution code *S*, you can specify some other character, for example, the vertical bar | or a blank:

```
      'C S<,|> F20.7' ⎕FMT NOS
123|456|789.1234567
 12|345|678.1234568
    123|456.1234560

      'C S<, > F20.7' ⎕FMT NOS
123 456 789.1234567
 12 345 678.1234568
    123 456.1234560
```

However, the spacing of the commas (or comma substitutes) supplied by qualifier *C* cannot be altered. If they appear, they are always inserted to separate groups of **three** digits. A different spacing (for example, breaking digits into clusters of two or four) could be achieved only with the picture format *G*.

Usually numbers represented in exponential form have only one digit to the left of the decimal, and so there wouldn't be any triplets of digits to separate by commas. But if you used leading zeros and had more than three of them, triplets of zeros could be separated by commas.

Substitutes for standard characters in *I*, *F*, and *E* formats

Certain characters which may occur in the result of these formats have standard meanings, but you can elect to substitute a different character for each of them. (In *G* format you can also elect to substitute characters for 9 and *Z*, which have special meanings in the picture used to control formatting.) The characters are:

* Field overflow character

0 Leading zero fill character

, Inserted character with *C* qualifier

. Decimal point (acceptable but never arises in *I* format)

E Character to mark exponent (acceptable anywhere but arises only in *E* format)

¯ Character to indicate negative exponent (acceptable anywhere, but does not apply to the sign of the number being represented, which is governed by the *M* decorator)

The decorator code *S* is followed by a string of characters enclosed in text delimiters. That string must have an even number of characters. The first member of each pair must be one of the standard characters for the format you're using. You may not show more than one substitution for the same standard character. The order in which the various pairs appear within the string doesn't matter.

In many European countries, the decimal point is marked not by a period but by a comma, and the period is used to separate triplets of digits in displaying a large number. For example, the number formed by adding one half to one million is written 1.000.000,5. That convention is achieved by substituting . for , and , for . , as follows:

```
      NOS←123456789.1234567  12345678.1234567  123456.1234567

      'C S<,..,> F 20.7' ⎕FMT NOS
 123.456.789,1234567
  12.345.678,1234568
     123.456,1234560
```

To illustrate some other substitutions, you might wish to use → to indicate an insignificant leading digit. That is illustrated in the following:

```
      'Z S<0→> C F 20.7' ⎕FMT NOS
→123,456,789.1234567
→→12,345,678.1234567
→→→→→123,456.1234567
```

When you're using *E* format, you may provide a substitute for the character *E* or for ¯ as the sign of the exponent:

```
      'S<E*¯↓> C E30.20' ⎕FMT NOS, ÷NOS,-NOS
    1.23456789123456700000*8
    1.23456781234567000000*7
    1.23456123456700000000*5
    8.10000006561000600000*↓9
    8.10000058320010000000*↓8
    8.10004374024210980000*↓6
   ¯8.10000006561000600000*↓9
   ¯8.10000058320010000000*↓8
   ¯8.10004374024210980000*↓6
```

Using substitute characters to insert dividing markers

To make a table easy to read, it is often useful to insert underlines to separate captions from the columns that they head, or to separate a column of figures from its total. You could first format the data without those, and then insert additional characters to provide the separators. But you could also achieve the same result by inserting certain numeric values into the data, and then, by substitution, causing those values to appear as the markers you want. The function *SUM*, shown below, illustrates an approach to preparing such a report with ⎕*FMT*.

```
    ∇ Z←YEARS SUM DATA; F1; F2
[1]   YEARS←YEARS,[0.5] 10000
[2]   DATA←DATA,[1] 1000000000,[0.5]+⌿DATA
[3]   F1←'X10, S<*¯> I4'
[4]   F2←'X3, S<0-*=> G<ZZZ,ZZZ,ZZ9>'
[5]   Z←(F1 ⎕FMT YEARS),[1] F2 ⎕FMT DATA
    ∇
```

The function takes two arguments: the vector of years that are to be the captions is the left argument, and the matrix of data to be reported is the right. The technique for generating the lines which separate the portions of the table is to insert into the data numbers too large to be represented, and then to substitute a suitable character for the stars that would otherwise appear in an overflow field. To prevent the overflow markers from running into each other, the *X* phrase (described in a later section) is used to insert blanks between the adjacent fields.

Statement 1 appends to *YEARS* a row containing the number 10000, too large to be represented in the format *F1*. Statement 2 appends to the matrix *DATA* a row containing the number 1 billion, and then a row containing the sums for each column. The value 1 billion is too large to be represented in the format *F2*.

Statement 3 contains the format control vector to be used for the captions. The overbar symbol ¯ is substituted for the star that would otherwise appear for an overflow field. The phrase *X10* provides 10 positions of blanks before each data field, so that the data in effect get 14 positions per field. Statement 4 contains the format control vector to be used for the body of the table. This time, the equal sign is used for the field overflow symbol, and serves to set off the the row of totals from the rest of the data. At the same time, a hyphen is substituted for the fill zero, and so appears at those positions marked by a 9 in the picture. Here is a data table *X*:

```
0     0  703  988
0   290  538  515
0   577    0    0
0   716    0    0
0     0  525    0
```

Here's a display of that data, for the years 1971+⍳4, using the function *SUM*:

```
      (1971+⍳4) SUM X
      1972        1973        1974        1975
      ----        ----        ----        ----
         -           -         703         988
         -         290         538         515
         -         577           -           -
         -         716           -           -
         -           -         525           -
==========  ==========  ==========  ==========
         -       1,583       1,766       1,503
```

The sign-dependent decorators *M N P Q*

These four decorators permit you to designate text that is to appear conditionally, depending on whether the sign of the number being represented is or is not negative. Each of them (if used) appears in the format phrase to the left of the key letter. Each is a single letter followed by whatever text you like, enclosed in text delimiters. The four decorator codes are:

	Leading	Trailing
Negative	*M*	*N*
Non-negative	*P*	*Q*

For example, if you would rather have negative values preceded by the minus sign, instead of the high negative sign otherwise used in APL, you can get that by the following:

```
      AMTS←100 ¯100 1250.85 ¯35.45

      'M<-> F10.2' ⎕FMT AMTS
 100.00
-100.00
1250.85
 -35.45
```

If you use **trailing** sign-dependent decorators, be aware that the characters **including the trailing decorators, if any** are positioned right justified in the field. So if you use a trailing decorator for numbers of one sign but not for the other, or if you use trailing decorators of different lengths, you'll throw off the alignment of the decimal points. To put the minus sign in trailing position in the preceding example would have the following effect:

```
      'N<-> F10.2' ⎕FMT AMTS
 100.00
¯100.00-
1250.85
 ¯35.45-
```

To preserve alignment, the trailing decorators (or the leading decorators, if you're preparing left justified displays) should be used in pairs, one for the negative and one for the non-negative values. And those pairs should have the same length. It's all right to provide **no** character as a substitute, by *M<>*. Here's what is probably a better way to write the preceding example:

```
      'M<> N<-> Q< > F 10.2' □FMT AMTS
  100.00
  100.00-
 1250.85
   35.45-
```

Decorator on both sides of the number

In some circumstances, you'd like negative numbers to have both leading and trailing markers. For instance, it is the practice in some kinds of accounting reports to show negative values enclosed in parentheses. That can be done by using both *M* and *N* decorators, and also by using *Q* to set a trailing decorator for the non-negative amounts that's the same size as the decorator used for the negative ones.

```
      'M<(> N<)> Q< > F 10.2' □FMT AMTS
  100.00
 (100.00)
 1250.85
  (35.45)
```

Floating dollar sign

In some situations, you'd like to print a currency symbol, for example a dollar sign, immediately to the left of the leftmost digit on each line. You'd like that whether the amounts are positive or negative. That's done by having the same symbol occur in both the *M* and the *P* text. For example:

```
      'M<$> P<$> N<-> Q< > F 10.2 ' □FMT AMTS
  $100.00
  $100.00-
 $1250.85
   $35.45-
```

Or the negative sign could be combined with the dollar sign in leading position:

```
      'M<-$> P<$> F 10.2' □FMT AMTS
   $100.00
  -$100.00
  $1250.85
   -$35.45
```

The text of the decorators can be as brief or as long as you want, provided there's room for that text within the total field width you've specified:

```
      'M<$> P<$> N< OVERDRAFT!> Q<            > F20.2' □FMT AMTS
  $100.00
  $100.00 OVERDRAFT!
 $1250.85
   $35.45 OVERDRAFT!
```

Deliberately aligning negative values differently

Since characters are normally placed right justified in the field, you could force negative numbers to be aligned differently from positive ones by selecting trailing decorators that differ markedly in length. Suppose, for example, that you'd like to have the positive values appear on the left side of the field, and the negative ones on the right. You can displace the positive ones 15 positions leftward by selecting a trailing decorator for the positive values that's 15 characters longer than the trailing decorator for negative values:

```
      'Q<                  > F  25.2' ⎕FMT AMTS
 100.00
                  ¯100.00
1250.85
                   ¯35.45
```

Leading decorators combined with leading zeros

If you use the codes *P* or *Q* to select a leading decoration text **and** at the same time use the leading zero qualifier *Z*, the effect is to put the leading decorator next to the left edge of the field. The numeric digits (and the trailing decorators, if any) are still aligned to the right edge. The space between them is filled with zeros. For example:

```
      'Z M<-> P<+> F 20.2' ⎕FMT AMTS
+0000000000000100.00
-0000000000000100.00
+0000000000001250.85
-0000000000000035.45
```

You'd get the same placement of the leading decorators even if you substituted a blank for the leading zero character:

```
      'Z S<0 > M<-> P<+> F 20.2' ⎕FMT AMTS
+              100.00
-              100.00
+             1250.85
-               35.45
```

Sign-dependent decorators with *E* format

If you use either of the trailing sign-dependent decorators *Q* or *N* with *E* format, the trailing text follows the four positions reserved for the letter *E*, the exponent, and the exponent's sign. If the trailing decorators differ in length, the units positions will no longer be aligned. For example, suppose *NOS* has the same value used in an earlier example:

```
      NOS←602880000000000000000000 ◇ NOS←NOS, 0, ÷NOS
```

```
'P<POS>Q<POS> M<NEGATIVE>N<NEGATIVE> E 30.3' ⎕FMT NOS, -NOS
          POS6.0E23 POS
          POS0.0E0  POS
          POS1.7E¯24POS
 NEGATIVE6.0E23 NEGATIVE
          POS0.0E0  POS
 NEGATIVE1.7E¯24NEGATIVE
```

Notice that the leading sign-dependent decorators do **not** apply to the sign of the exponent, but only to the sign of the number being represented. However, you can substitute a single character either for *E* or for ¯ in the exponent. For example:

```
'S<E*¯-> P<POS>Q<POS> M<NEGATIVE>N<NEGATIVE> E30.2' ⎕FMT NOS, -NOS
          POS6.0*23 POS
          POS0.0*0  POS
          POS1.7*-24POS
 NEGATIVE6.0*23 NEGATIVE
          POS0.0*0  POS
 NEGATIVE1.7*-24NEGATIVE
```

Background text with *R*

The code *R* permits you to specify a background text which shows through in any positions not occupied by the digits or decorators. The unused spaces can only be at the left side of the field (or on the right, if you use the left justifying qualifier *L*). When code *R* is combined with *E* format, the background text can fill unused positions at the left; it can't fill the positions left blank when the exponent has no negative sign or requires only one digit.

If you use the leading zero qualifier *Z*, or if the particular numbers and decorators fill the field, the background text won't find any unused positions at all, and so won't appear. In the following example, the background consists of all 26 letters of the alphabet, and the field is 26 characters wide. That's intended to make it easier to see how the background text is aligned in the field.

```
     'R<ABCDEFGHIJKLMNOPQRSTUVWXYZ> F 26.2' ⎕FMT AMTS
ABCDEFGHIJKLMNOPQRST100.00
ABCDEFGHIJKLMNOPQRS¯100.00
ABCDEFGHIJKLMNOPQRS1250.85
ABCDEFGHIJKLMNOPQRST¯35.45

     'L R<ABCDEFGHIJKLMNOPQRSTUVWXYZ> F 26.2' ⎕FMT AMTS
100.00GHIJKLMNOPQRSTUVWXYZ
¯100.00HIJKLMNOPQRSTUVWXYZ
1250.85HIJKLMNOPQRSTUVWXYZ
¯35.45GHIJKLMNOPQRSTUVWXYZ
```

If your decorator text includes blanks, or if you substitute blank for the leading zero character, the background does **not** show through those blanks.

If the length of the background text you supply is different from the width of the field, the background text is truncated or repeated as necessary to match the field width. In particular, if the background is a single phrase or a single character, it is repeated throughout.

Here's an example in which a background consists of a dot and a space, as you might use in laying out a table of contents:

```
      NOS←1 45 180 762 1101

      'R<. > P< >I4O' ⎕FMT NOS
. . . . . . . . . . . . . . . . . . . . .  1
. . . . . . . . . . . . . . . . . . . . . 45
. . . . . . . . . . . . . . . . . . . . . 180
. . . . . . . . . . . . . . . . . . . . . 762
. . . . . . . . . . . . . . . . . . . . . 1101
```

Here's an example in which a single star is used as a **cheque protection character:**

```
      AMT←45 16.77 2045.7

      '<THE SUM OF >, R<*> P<$> F 15.2' ⎕FMT AMT
THE SUM OF *********$45.00
THE SUM OF *********$16.77
THE SUM OF *******$2045.70
```

Background text to mark zero values

You can combine the background decorator *R* with the qualifier *B* (which leaves blank the field used to represent a number whose value is zero) in such a way that the background text shows through only for the components whose value is zero.

For example, suppose you would prefer to print the word *NIL* for any value that is zero. You could do that in the following way:

1. Choose a format so that the numbers that **are** printed require at least as many print positions as you plan to use for the message. In this example, since *NIL* has three letters, pick a format such as *F*10.1, which will always give you one place after the point, the point itself, and at least one place to the left of the point.

2. Use the qualifier *B* so that zero values are left blank.

3. Use the decorator *R* to insert the message that will show through when the field is left blank. Note that the background text will need enough leading blanks so that the message itself appears only in the rightmost positions, where it will always be overwritten by the digits **except** when the entire field is made blank because of a zero value.

Here is an array *X*, several of whose components are zero, formatted with 1 decimal place:

```
      'F 10.1' ⎕FMT X
 0.0      81.4      89.8       0.0      76.1
69.5      45.2      26.4      66.6      11.7
21.0      11.4      54.7       0.7      17.6
60.0       0.0       8.3       0.0      37.6
```

Here is that array, formatted so that the zeros are represented by the word *NIL*:

```
      'R<       NIL> B F 10.1' ⎕FMT X
 NIL      81.4      89.8       NIL      76.1
69.5      45.2      26.4      66.6      11.7
21.0      11.4      54.7       0.7      17.6
60.0       NIL       8.3       NIL      37.6
```

Notice that the *R* text has seven leading blanks, to assure the the letters *NIL* appear at the right edge of the ten-position fields.

Formatting characters with ⎕*FMT* code *A*

The key letter *A* is used to format character data. It is the simplest of all the formats. The only item to the right of the key letter *A* is a number indicating how wide the field is to be. The column of characters being formatted appears at the right edge of the field, and the rest of the field is blank. For example, suppose *C* is a 3-by-2 character matrix as follows:

```
      C
↑X
*Y
↓Z
```

You can format those two columns with two phrases (or with a single phrase that applies to both). Here's a statement containing two slightly different format phrases. The first provides a field 3 positions wide in which to write the first column of data. Since the data appear on the right, the first two positions are blank. The second phrase provides a field 2 positions wide, so that you get blanks in the first position and the data in the second.

```
      'A3, A2' ⎕FMT C
  ↑ X
  * Y
  ↓ Z
```

If you want the characters reproduced the way they were, then you want fields one position wide for each of the columns of the data:

```
      'A1' ⎕FMT C
↑X
*Y
↓Z
```

The example just shown works because the single phrase *A1* (which describes how to format **one** column of the argument) is reused for each of the columns to be formatted. The single phrase can be successfully reused for all columns only because it's the only format phrase, and it's appropriate to all columns of the data. But if there were numeric columns to be formatted as well, re-using *'A1'* wouldn't work. So a more general form for formatting character data is

```
      'nA1' ⎕FMT C
```

in which you substitute for n the number of columns of characters to be formatted. For example, since *C* has two columns,

```
      '2A1' ⎕FMT C
↑X
*Y
↓Z
```

Using *A* format to display a vector

In all formats, the *A* format included, ⎕*FMT* treats a vector of data as a one-column matrix. For example:

```
      'A3' ⎕FMT 'ABCD'
A
B
C
D
```

Watch out: If your character vector is a fragment of text (for example, somebody's name) and you want it to appear horizontally, the following is **not** the way to do it:

```
      NAME←'JACK'

      'A1' ⎕FMT NAME
J
A
C
K
```

If you want the name to appear as a horizontal caption, you need first to make it into a 1-row matrix:

```
      'A1' ⎕FMT 1 4⍴'JACK'
JACK
```

You may find it useful (here and elsewhere) to employ a function that turns any array into a matrix, such as the following one:

```
    ∇ Z←MATRIX X
[1]   Z←((×/¯1↓⍴X),×/(¯1+⍴⍴X)↓⍴X)⍴X
    ∇
```

With something like that, the following example becomes possible:

```
      YEARS←1975 1976 1977
      SALES←18450 19220 21680
      SALESMAN←10↑'JACK JONES'

      '10A1, I5, C F10.2' ⎕FMT (MATRIX SALESMAN; YEARS; SALES)
JACK JONES 1975  18,450.00
           1976  19,220.00
           1977  21,680.00
```

When you're using □*FMT* to format character data, the important thing to remember is that □*FMT* requires a field for **each column** of the data. That remains true regardless of whether the data being formatted are columns of numbers or columns of characters. The point is important, because with character data you are likely to be treating a block of adjacent columns as part of a single word.

Another way to look at it is this: For the *A* format, put to the left of the *A* the repetition factor showing how many columns of characters are being processed, and to the right the spacing factor. Then the phrase

```
10A3
```

means that 10 columns of character data should be printed triple spaced (and so will occupy 30 print positions).

Here's an example in which the character data is a matrix *NAMES*, each row of which represents the name of a part. There is also a vector of part numbers *NOS*, and a matrix *STOCK* containing in one column the number of parts in stock and in the second column the unit price.

```
      NAMES
CRANK
HANDLE
DOOR HINGE

      NOS
14838 2007623 17

      STOCK
 190       4.22
  18       1.9
4000       0.17

      '10A1, G< 999-999-99>, C I8, F7.2' □FMT (NAMES;NOS;STOCK)
CRANK      000-148-38     190   4.22
HANDLE     020-076-23      18   1.90
DOOR HINGE 000-000-17   4,000   0.17
```

If you want a block of text simply reproduced unchanged in the result, you can also do that directly, without using □*FMT* for that part of the result. For example, the foregoing example could have been handled just as well with an instruction in which the matrix *NAMES* is taken as is, and only the numeric data are formatted:

```
      NAMES, 'G< 999-999-99 > , C I8, F7.2' □FMT (NOS; STOCK)
CRANK      000-148-38     190   4.22
HANDLE     020-076-23      18   1.90
DOOR HINGE 000-000-17   4,000   0.17
```

In the preceding example, *NOS* and *STOCK* are both numeric, and have the same number of rows. You could achieve the same display by catenating them. In that case, the parentheses wouldn't be necessary:

```
      NAMES, 'G< 999-999-99>, C I8, F7.2' □FMT NOS,STOCK
```

Background text in the spaces between characters

If the number to the right of the key letter *A* is greater than one (for example, three) you are assigning three print positions to contain one column of characters. The characters appear right justified, so the preceding print positions in each field are blank. You can use the *R* decorator to supply a background text to fill those blank positions. For example:

```
      NAME←1 12⍴'HYMAN KAPLAN'

      'R<*> A2' ⎕FMT NAME
*H*Y*M*A*N* *K*A*P*L*A*N
```

But notice that the background text is prepared independently for each field, and doesn't span the blanks in several fields:

```
      'R<1234567890> A 3' ⎕FMT NAME
12H12Y12M12A12N12 12K12A12P12L12A12N
```

Other qualifiers or decorators with *A* format

There aren't any. Not even *L* has any effect on a character field.

Data of the opposite type

If you use *A* format to print numeric data, you get that field filled with stars:

```
      'A1' ⎕FMT ⍳3
*
*
*
```

This is an effect you sometimes may get unintentionally. The usual cause is a mis-match between the format phrases you're using and the columns of data you've supplied. The preceding example involving the 17-column matrix *NAMES*. Iif you had specified 18*A*1 where 17*A*1 was needed, you would be attempting to use *A* format on the eighteenth column, which (in this example) is not part of *NAMES* at all, but the first column of the numeric array *X*. The first column of *X* therefore appears as a single column of stars. Moreover, the omitted column is no longer the second column of *X* but the third:

```
      '18A1, I6, I0, F8.2' ⎕FMT (NAMES; X)
COVER NUT LOCKING*    16
GASKET    RUBBER *    45
CAM SHAFT STEEL  *   202
```

Notice that there's now no column of *X* left for the phrase *F*8.2 to format.

The same principle holds if you attempt to apply to character data any of the formats intended for representation of numeric data. Suppose *NAMES* is a 3-by-17 character matrix, but you specify '*I*1' rather than '*A*1':

```
      'I1' ⎕FMT NAMES
****************
****************
****************
```

Were you to ask for *'I8' ⎕FMT NAMES*, you'd get each column of *NAMES* represented by a field of stars 8 print positions wide.... that's a 3-by-136 matrix of stars, 8 stars for each of the 17 columns in *NAMES*—a result too wide to be illustrated here.

Inserting a constant text

You can insert into the result a field (for example, a label) which does not depend on the data being formatted, but is simply reproduced on each row of the result. (The *G* format also permits you to do that.)

A format phrase containing **no** key letter, but consisting solely of a string of text enclosed in text delimiters, causes that text to be reproduced in the result. Assuming that the variables *PARTNAME*, *NOS*, and *STOCK* have the same meaning as in the preceding example, additional labelling could be provided like this:

```
      F
<PART NAME: >,10A1,<  NO:>,G< 999+999+99>,<  STOCK:>,CI6,<  PRICE:>,F5.2

      F ⎕FMT (PARTNAME; NOS; STOCK)
PART NAME: CRANK       NO: 000-148-38  STOCK:   190  PRICE: 4.22
PART NAME: HANDLE      NO: 020-076-23  STOCK:    18  PRICE: 1.90
PART NAME: DOOR HINGE  NO: 000-000-17  STOCK: 4,000  PRICE: 0.17
```

Constant text that appears after the last column of data

The system formats each column of the right argument, using the formatting phrases from the left as necessary until all the columns on the right have been dealt with. If, when the last column has been formatted, there's still one or more phrases remaining to the right of the last-used phrase, what happens? The rule is this:

1. If the next formatting-phrase is one that requires a column of data (but there are no more columns of data), quit.
2. If the next phrase is a constant text (and thus doesn't require a column of data) **and** is enclosed in parentheses, quit.
3. If the next phrase is a constant text and is **not** enclosed in parentheses, include the text as the last field of the result, and then quit.

To illustrate this, consider a format phrase that includes provision for formatting two columns of numbers, but has in addition three constant text fields:

```
      F←'< LEFT>, I6, < MIDDLE>, I6, < RIGHT>'
```

First apply that to data with two columns:

```
      F ⎕FMT (⍳3; 10×⍳3)
LEFT      1 MIDDLE     10 RIGHT
LEFT      2 MIDDLE     20 RIGHT
LEFT      3 MIDDLE     30 RIGHT
```

Now apply the same format statements to a single column of data:

```
      F ⎕FMT ⍳3
LEFT      1 MIDDLE
LEFT      2 MIDDLE
LEFT      3 MIDDLE
```

If you supply three columns of data, some of the formatting phrases must be reused:

```
      F ⎕FMT (⍳3; 10×⍳3; 100×⍳3)
LEFT      1 MIDDLE     10 RIGHT LEFT    100 MIDDLE
LEFT      2 MIDDLE     20 RIGHT LEFT    200 MIDDLE
LEFT      3 MIDDLE     30 RIGHT LEFT    300 MIDDLE
```

However, if *F* is changed so that the text *MIDDLE* is enclosed in parentheses, then that text field appears **only if there is a data field further to the right:**

```
      F2←'< LEFT>, I6, (< MIDDLE>), I6, < RIGHT>'

      F2 ⎕FMT ⍳3
LEFT      1
LEFT      2
LEFT      3

      F2 ⎕FMT (⍳3; 10×⍳3; 100×⍳3)
LEFT      1 MIDDLE     10 RIGHT LEFT    100
LEFT      2 MIDDLE     20 RIGHT LEFT    200
LEFT      3 MIDDLE     30 RIGHT LEFT    300
```

Empty fields

With ⎕*FMT* (but not with ⍕) it is acceptable to use a formatting phrase which says that a column of data is to be represented in **zero** print positions (that is, in a field whose width is 0). In that case, of course, you won't see those characters. The effect is equivalent to omitting that column from the result. For example, suppose *X* is a matrix with three columns, like this:

```
18002  16 1.17
18003  45 3.60
18009 202  .77
```

To show just the first and third columns of *X*, omitting the second, you could compress that column out of the data:

```
      'I6, F 8.2' ⎕FMT 1 0 1/X
18002    1.17
18003    3.60
18009    0.77
```

You could achieve the same result by leaving the right argument unchanged, but assigning 0 width to represent the second column:

```
      'I6, I0, F 8.2' ⎕FMT X
 18002    1.17
 18003    3.60
 18009    0.77
```

The same technique could be used to print only part of a table of character data. Suppose *NAMES* is an array of part names and has 17 columns:

```
      NAMES
COVER NUT LOCKING
GASKET    RUBBER
CAM SHAFT STEEL
```

Suppose *X* is data describing the items named by *NAMES*. In a report you want to print only the first 10 columns of *NAMES*. You can do that by using ⎕*FMT* with a left argument like this:

```
      '10A1, 7A0, I6, I0, F8.2' ⎕FMT (NAMES; X)
COVER NUT  18002    1.17
GASKET     18003    3.60
CAM SHAFT  18009    0.77
```

If for some reason you wanted to represent *NAMES* just by the characters at positions 1 and 11 within it, you could assign 0 width to the the unwanted positions, like this:

```
      'A1, 9A0, A1, 6A0, I6, I0, F8.2' ⎕FMT (NAMES; X)
CL 18002    1.17
GR 18003    3.60
CS 18009    0.77
```

Moving to a new position in the display with phrases *X* and *T*

The system constructs the result of ⎕*FMT* one field at a time, processing the columns of data in the order that they are stated in the right argument, from left to right. At any time during that process, you can direct that the next field should start **not** at the next position to the right of the position most recently written, but at some other print position.

The key letter *X* lets you describe the new position as a departure from that "next" position. The phrase *X*5 means "now move 5 positions further to the right." The phrase *X*¯5 or *X*-5 means "Now move five positions to the left."

The key letter *T* lets you give the same sort of instruction, but instead of stating the skip as a departure from what would otherwise have been the next position, *T* requires you to state the absolute position number of the print position next to be formatted. Thus, the phrase *T*22 means that the first column of the next field is to be at position 22. That is, that there should be 21 positions further to the left of it (since ⎕*FMT* refers to the printing positions in 1-origin indexing).

If your *X* or *T* instruction calls for the next field to start at a position further to the right than the rightmost position yet formatted, the space between is filled with blanks. Consider the spacing of the two fields in this example:

```
      'I3, C Z I11' ⎕FMT (⍳3; ⍳3)
1000,000,001
2000,000,002
3000,000,003
```

The second field (because of the use of the qualifier *Z*) contains a printed character in every position, and comes right up against the first field. Suppose you'd prefer to displace the second field five positions to the right. This does it:

```
      'I3, X5, C Z I11' ⎕FMT (⍳3; ⍳3)
1      000,000,001
2      000,000,002
3      000,000,003
```

The same effect could be produced by specifying that the second field is to begin in position 9 (that is, after leaving 3 positions for the first field, and 5 positions blank):

```
      'I3, T9, C Z I11' ⎕FMT (⍳3; ⍳3)
1      000,000,001
2      000,000,002
3      000,000,003
```

Moving back over the fields already written

The result of ⎕*FMT* is constructed one field at a time, processing the columns of data in the order that they are stated in the right argument, from left to right.

If the displacement in an *X* phrase is negative, or if the destination in a *T* phrase is further to the left than the last position completed, the next field **overwrites the positions already formatted.** You could use some such device to move back and insert a field in a sequence different from its position in the right argument. Consider a very simple example: writing the numbers 1 2 3 in one field, and 4 5 6 in another. A simple way to do this might be to give each a field 10 positions wide:

```
      'I10, I10' ⎕FMT (1 2 3; 4 5 6)
         1          4
         2          5
         3          6
```

But if you want to have the second column displayed to the left of the first one, you could do it by the following:

1. After the phrase that formats the first column, use the phrase *X* or *T* to specify that the next field starts at a position closer to the left edge.

2. Make the second field narrow, so that it fits into the available space in the first field.

Here's an instruction to do that:

```
      'I10, X-9, I3' ⎕FMT (1 2 3; 4 5 6)
4       1
5       2
6       3
!
```

The symbol ! shows the position last to be formatted. If your instruction were now to ask for another field, that field would start at the next position to the right of the ! —unless you once more used *X* or *T* to skip to a new position.

Overwriting one field with another

The preceding example shows the second field (containing the numbers 4 5 6) overwriting a portion of the first field. In that case, the first field had been left with lots of blanks at the left. Because the positions thus overwritten previously contained blanks, nothing much was lost by overwriting them. But you can, if you want, overwrite text or digits in the previously formatted fields. If you do that, the following rule applies:

> Each printing character of the new field replaces a character of the old field, **but** a fill blank in the new field leaves unchanged the character previously written at that position.

To illustrate that rule, suppose you're formatting a matrix that has two columns. Numbers in the first column are in the millions, but numbers in the second column are much smaller:

```
      X
1000000    ¯111
2000000     ¯22
3000000      ¯3
```

Formatting *X* in fields 11 positions wide might give you this:

```
      'C I11' ⎕FMT X
1,000,000       ¯111
2,000,000        ¯22
3,000,000         ¯3
```

By inserting a phrase *X*-7, you'd move the second field 7 positions to the left, like this:

```
      'C I11, X-7' ⎕FMT X
1,000,000¯111
2,000,000 ¯22
3,000,000  ¯3
```

The first seven positions of the second field now overlap the last 7 positions of the first field. But because the first positions of the second field are all blanks, digits from the first field are still visible. If the reposition is by ¯10 rather than ¯7, characters from the second field overwrite characters from the first:

```
      'C I11, X-10' ⎕FMT X
1,000,¯111
2,000,0¯22
3,000,00¯3
```

When one field overwrites another, only **fill** blanks let a previously written character show through. Fill blanks are blanks used to pad the left side of a field that is not filled (or the right side, if you're using qualifier *L* to get left justification), or the blanks on a line that has zero value when you're using the qualifier *B*. Blanks that are explicitly part of a decorator, or are written as explicit substitutes for some standard character, don't let the earlier characters show through, and overwrite them. For example, if negative values in the second column are marked with a leading decorator consisting of a plus sign with a blank on either side of it, those blanks **do** overwrite characters written earlier:

```
      'C I11, X-10, M< - > I11' ⎕FMT X
1,00 - 111
2,000 - 22
3,000, - 3
```

Skipping to the end of the positions already formatted

After skipping back to insert a field on top of some of the fields already formatted, you may want to start the following field at the next position to the right of all those so far prepared. You can do this by keeping track of where you now are, and supplying the appropriate number with code *X* or *T*.

Alternatively, to make that particular job simpler, you can use *T* alone in a phrase, with no number following it. That means that the next field should begin at the next position to the right of the rightmost one so far constructed. For example:

```
      'C I11, X-10, I11, T, I3' ⎕FMT (X; 777 888 999)
1,000,¯111777
2,000,0¯22888
3,000,00¯3999
```

APPENDIX

CONDENSED SUMMARIES OF FORMATTING RULES

1. SHAPE AND CONFORMABILITY

Monadic ⍕: *Z←⍕X*

Argument: An array, character or numeric, of any rank.

Result: For a character argument, the argument unchanged.

For a numeric argument, a character array, in general having the same rank as the argument, except that a scalar argument yields a vector result. The result has the same length as the argument in each axis other than the last. The result contains the same characters as an unformatted display, and is dependent on ⎕*PP*.

Dyadic ⍕: *Z←F⍕X*

Right argument: A numeric array of any rank.

Left argument: A numeric vector or scalar. The vector consists of **pairs** of integers. Within each pair, the first integer, w, is field width, the second, **d**, digits. A scalar or one-element left argument *F* is treated as `¯2↑`*F*. The number of pairs in *F* must divide evenly into the length of the last axis of *X*. That is, `0=(2⌈⍴,`*F*`)|¯1↑⍴`*X*.

Result: A character array of the same rank as the argument, except that a scalar right argument yields a vector result. The result has the same length as the right argument along each axis other than the last.

Dyadic ⎕*FMT*: *Z←F* ⎕*FMT X* **or** *Z←F* ⎕*FMT* `(`*X*`;`*V*`;`*W*`)`

Right argument: An array, either character or numeric, having rank 2 or less. A vector or scalar is treated as a one-column matrix.

Alternatively, the right argument may be a **list** of such arrays, enclosed in parentheses, with its members separated by semicolons.

Left argument: A character vector, consisting of **phrases** separated by commas. There is no requirement that the number of phrases conform to the number of columns in the right argument.

Result: A character matrix having as many rows as the right argument, or, if the right argument is a list, as the array having the greatest number of rows. The width of each field within the result is controlled solely by the corresponding phrase from the left argument; the total width of the result depends on the content of those phrases, and the number of columns in the right argument.

2. STRUCTURE AND MEANING OF CONTROL PAIRS IN THE LEFT ARGUMENT OF ⍕

Structure: The left argument *F* is treated as `(-2⌈⍴F)↑F`. The first member of each pair, **w**, controls field width, and the second, **d**, digits.

Meaning: For all fields controlled by a given pair:

w: The total width of the field (including unused space appearing as leading or trailing blanks).

w zero: Field width is unspecified, and depends upon the actual values in the corresponding column(s) of the right argument. **w** is treated as 1 more than the maximum number of print positions needed to represent any number in the column or columns controlled by the pair.

w larger than needed: With ⍕, formatting is always right justified, so unused positions appear as blanks at the left side of the field.

w insufficient: If the number of positions required to represent a number is greater than w, that row of the field is filled with **w** stars. ⍕ does not provide a substitute for `*`.

d zero or positive: Fixed-point representation with **d** decimal places. Trailing zeros appear.

d not zero: Decimal point appears (in either fixed or exponential format). Decimal points are aligned.

d negative: Exponential representation with at least `|d` digits.

d excessive: If **d** exceeds 18, the additional low order positions are `0`.

X **negative:** Negative values in the right argument are marked by `¯` in the position to the left of the high order digit. ⍕ does not provide a substitute for `¯`.

3. STRUCTURE OF □*FMT* CONTROL VECTOR

Literal text: Scanning across the entire control vector from left to right, literal text is the sequence of characters enclosed by:

< until next >
⊂ until next ⊃
¨ until next ¨
□ until next □
▯ until next ▯

Blanks: Outside sequences of literal text, blanks are not required, and if supplied have no effect.

Separation into phrases: The left argument of □*FMT* is divided into phrases, separated by commas (other than those appearing within literal text).

Repetition of phrases: A non-negative integer at the start of a phrase means the phrase is repeated that many times. A non-negative integer followed by several phrases enclosed in parentheses means the entire sequence in parentheses is repeated that many times:

The following are equivalent:
3(aaa,bbb)
aaa,bbb,aaa,bbb,aaa,bbb

Last phrase used: The phrases in the left argument *F* are used in turn to format the columns of the right argument *X*, and reused as necessary if there are more columns than phrases. Formatting is complete when

1. The last column of *X* has been formatted

and

2. There is no phrase in *F* to the right of the phrase last used

or

The next phrase in *F* applies to a column of data (i.e., uses key letters *G I F E A*)

or

The next phrase in *F* is a constant text not enclosed in parentheses.

Type of data: Whenever numeric data is formatted with the character format *A*, or character data is formatted with the numeric formats *G I F E*, the result field is treated in the same way as field overflow. That is, that row, for the entire width of the field, is filled with *s (or with the character substituted for *).

4. COMPONENTS OF □*FMT* PHRASE

KEY LETTER ↓

Optional, appearing to left of key letter, in any order — Required

Decorators	Qualifiers		
←----------→	←--------→		←--------→
M<..> *N*<..> *P*<..> *Q*<..> *R*<..> *S*<αω>	*B K*n	*G*	<picture>
M<..> *N*<..> *P*<..> *Q*<..> *R*<..> *S*<αω>	*L C Z B K*n	*I*	w
M<..> *N*<..> *P*<..> *Q*<..> *R*<..> *S*<αω>	*L C Z B K*n	*F*	w.d
M<..> *N*<..> *P*<..> *Q*<..> *R*<..> *S*<αω>	*L C Z B K*n	*E*	w.d
R<..>		*A*	w

For .. supply appropriate decorator text.
For picture supply appropriate picture text, with digit positions marked *Z* or 9.
For αω supply pairs of character substitutions.
For n supply power of 10 for scale factor.
For w supply field width.
For d supply number of decimals or significant digits.

5. EFFECTS OF □*FMT* QUALIFIERS

K: Scale factor

Applicable formats: *G I F E*.

Specified by: Letter *K* followed (optionally, by a negative or minus sign) and by digits representing an integer, anywhere within the phrase to the left of the key letter.

Effect: Before formatting, multiplies that column of data by the power of 10 shown to the right of the *K*. Since *G* and *I* formats represent numbers to the nearest integer, *K* qualifier permits representation of digits beyond the decimal point that would otherwise be lost.

B: **Blank zero values**

Applicable formats: *G I F E.*

Specified by: Letter *B* within the phrase anywhere to the left of the key letter.

Effect: Representation of a number whose representation contains no significant digits is left blank. Optional sign-dependent decorators or commas do **not** appear. If qualifier *Z* is also used, leading zeros are **not** printed. Background text, if any, shows throughout that row of the field.

Z: **Leading zeros**

Applicable formats: *I F E.*

Specified by: Letter *Z* within the phrase anywhere to the left of the key letter.

Effect: All available positions to the left of the high order digit are filled with zeros. If a leading decorator (*M* or *P*) is used, the decorator text is left justified, and the zeros fill the positions (if any) between the last character of the decorator and the high order digit.

If background text is also specified (decorator *R*), no positions remain at which background can appear. If blank zero value (qualifier *B*) is also set, leading zeros do **not** appear. If the representation is left justified (qualifier *L*), leading zeros do **not** appear.

C: **Commas**

Applicable formats: *I F E.*

Specified by: Letter *C* within the phrase anywhere to the left of the key letter.

Effect: Digits to the left of the decimal point are separated in groups of three by commas. If leading zeros are also requested (qualifier *Z*), commas appear at **all** eligible positions (even if the leftmost position in the field). But in the absence of qualifier *Z*, a comma appears only if a digit generated as part of the number's representation appears on **both** sides of it. A comma may thus be overwritten by a leading sign-dependent decorator (*P* or *M*) or by background text (decorator *R*).

L: **Left justify**

Applicable formats: *I F E.*

Specified by: Letter *L* within the phrase anywhere to the left of the key letter.

Effect: The number representation is built from the left side of the field. At the left edge appear leading sign-dependent decorators (if any), followed immediately by the high order digits of the number. Alignment of decimal points is not preserved.

If qualifier *C* is also used, commas are inserted to break the digits into groups of three counting leftward from the units position in the usual way; vertical alignment of commas is not preserved. Leading zeros do not appear, even if qualifier *Z* is also used. Trailing decorators appear immediately to the right of the rightmost position reserved for the number-representation. Any positions further to the right are filled with blanks. If a background decorator *R* is used, the background text shows through only at the trailing positions on the right.

6. EFFECTS OF □*FMT* DECORATORS

P and *M*: Leading sign-dependent decorators

Applicable formats: *G I F E.*

Specified by: Letter *P* or *M* followed by the decorator text enclosed in text delimiters, anywhere within the phrase to the left of the key letter.

Effect: The decorator appears to the left of the number representation of numbers that are positive or zero (decorator *P*) or are negative (decorator *M*).

In format *G*, the decorator appears to the left of the leftmost 9 in the picture, and to the left of the high order digit; it may overwrite portions of the picture or positions marked *Z*.

In formats *I* and *F*, the decorator appears immediately to the left of the high order digit. However, in the presence of qualifiers *Z* or *L*, it appears at the left edge of the field. With qualifier *L*, the digits of the number representation are displaced rightward by the length of the decorator.

Q and *N*: Trailing sign-dependent decorators

Applicable formats: *G I F E.*

Specified by: Letter *Q* or *N* followed by the decorator text enclosed in text delimiters, anywhere within the phrase to the left of the key letter.

Effect: The decorator appears immediately to the right of the rightmost position reserved for the representation of numbers that are positive or zero (decorator *Q*) or negative (decorator *N*). (The rightmost position ordinarily contains the low order digit, but may contain an underline or a blank if the requested number of digits exceeds system precision.)

In format *G*, a trailing decorator appears to the right of the rightmost 9, overwriting the picture or positions marked *Z*.

In formats *I* and *F*, provided qualifier *L* is not used, a trailing decorator appears right justified, and displaces the number representation leftward by the length of the decorator. If only one of the trailing decorators is used, or they differ in length, the alignment of decimal points is lost.

R: Background text

Applicable formats: *G I F E A.*

Specified by: Letter *R* followed by the text of the background decorator enclosed in text delimiters, within the phrase anywhere to the left of the key letter.

Effect: The background text is truncated or repeated as necessary to fill the entire width of the field. Characters from the background text show through in the result only at positions otherwise left blank.

In format *G*, the background text shows through only at positions lying between the leftmost and rightmost positions marked *Z*, and further to the left than the leftmost 9, or further to the right than the rightmost 9, and not used to print a digit of the number representation.

In formats *I*, *F*, and *E*, the background text shows through only at the left side of the field, to the left of the high order digit and any leading sign-dependent decorator, or (if qualifier *L* is used) to the right of the rightmost position reserved to represent the number and to the right of the trailing sign-dependent decorator. If qualifier *Z* is used but not qualifier *L*, there remain no positions at which the background text can appear. If qualifier *B* is used and the value being formatted is zero, neither the number representation nor any sign-dependent decorators appear, and the background text occupies the entire field.

S: Substitution of standard characters

Applicable formats: *G I F E.* The permissible substitutions depend on the format with which they are used:

G:	9	*Z*	0	*				
I:			0	*	,			
F:			0	*	,	.		
E:			0	*	,	.	*E*	−

Specified by: The letter *S*, followed by an even number of characters enclosed in text delimiters, in which the first member of each pair is a standard character and the second a substitute for it.

Effect: In format *G*, the signficance of 9 or *Z* in the picture is assigned to the substitute character; the same substitute may not be assigned to both.

In the other formats, the substitute character appears instead where the display would otherwise have contained:

0 as leading zero
* as field overflow marker
, as separator between triplets
. as decimal point
E as exponent (in exponential format only)
− as negative sign of exponent (in exponential format only)

INDEX

In this index, names beginning with the character ⎕ are alphabetized as if spelled *QUAD*. An entry whose first letter is a capital indicates that the entry is the title of a section.

A

A, character-formatting phrase, 47 60
absolute reposition, 53
aligning negative values differently from non-negative, 44
Alignment of columns, in ⎕*FMT* formats *I*, *F*, and *E*, 35
appropriate field width for decimal displays, 14

B

B, blank-for-zero qualifier, 37 61
background text, summary of rules regarding, 63
Background text in the space between characters, 50
Background text to fill *Z* positions left blank, in picture format, 30
Background text to mark zero values, 46
Background text with *R*, 45
blank, to represent zero value, 37
Blank for zero value with *B*, 37
blank positions in display, inserted by repositioning further to the right, 54
both sides, sign-dependent decorator on, 43

C

C, comma-insertion qualifier, 38 61
Capturing the way the system displays numbers, 8
character data, monadic thorn applied to, 10
  formatting with ⎕*FMT* code *A*, 47
  columns of, 49
  background text intermingled with, 50
character data formatted with numeric code, 50
character vector, treated as one-column matrix by ⎕*FMT*, 20
cheque protection character, 46
columns, alignment of, 35
"columns" and "fields," note about, 5
comma, to separate phrases within format control vector, 21
  substitute for, 39
commas, insertion of, 38 61
Components of ⎕*FMT* phrase, 60
conformability and shape rules, summary of, 57
constant text, insertion of, 51 59
Constant text that appears after the last column of data, 51
control pair, one applied to all columns, 12
control pairs, number of, 11
control pairs in left argument of ⍕, summary of rules regarding, 58
control vector, left argument of ⎕*FMT*, 20
  ⎕*FMT*, structure of, 59

D

Data of the opposite type, 50
data to be formatted by ⎕*FMT*, 19
data type, summary of rules regarding mis-match of data type to format phrase, 60
Decimal and exponential fields in the same display, 17
decimal displays, appropriate field width for, 14
decimal format with dyadic thorn, 11
decimal or exponential form, selection of, with dyadic thorn, 11
decimal point, 31
  substitute for, 39
Decimal point in picture format, 26
Decorator on both sides of number, 43
decorators, position of, in picture format, 27
  sign-dependent, with picture format, 28
  leading, combined with leading zeros, 44
  others with *A* format, absence of, 50
decorators and qualifiers, list of, 24 25 60
decorators and qualifiers, and the formats with which they are available, 24 25 60
default display, system's, captured with monadic thorn, 8
delimiters, text, 21 59
digits, more requested than system can provide, 7 18
  significant, 8
dividing marks, inserted by use of substitute for field overflow, 41
dollar sign, floating, 43

Dyadic thorn: controlling field width and precision, 10

E

E, format phrase, summary of options with, 60
E character, substitute for, 39
E format, see also exponential format
Empty fields, 52
exponential and decimal fields in the same display, 17
exponential form, representing numbers in, 15
with dyadic thorn, 16
minimum field width for, 17
exponential format, sign-dependent decorators with, 44
Exponential format *E*, 33
exponential or decimal form, selection of, with dyadic thorn, 11

F

F, format phrase, summary of options with, 11 31 60
F format, see also fixed-point decimal format, □*FMT*
field, width of, in result of monadic ⍕, 9
empty, with □*FMT*, 52
field overflow, 6 18 32
field overflow character, substitute for, 39
Field overflow with picture format, 29
field too small, 6
field wider than needed, 6
field width, appropriate, for decimal displays, 14
field width and precision, how controlled in dyadic use of thorn, 10
Field width 0 lets the system pick the width, 12
"fields" and "columns," note about, 5
Fixed-point decimal format with dyadic thorn, 11
Fixed-point format *F*, 32
Floating dollar sign in picture format, 28
Floating dollar sign, 43
format control vector, 20
format phrase, structure of, 23
"right argument" of, 23
"left argument" of, 24
format phrases, types of, with □*FMT*, 2
list of, 3
format primitives, purpose of, 1
summary of, 2
formats with which qualifiers and decorators are available, 25
Formatting characters with □*FMT* code *A*, 47
Formatting several arrays when some require more rows than others, 20

G

G, format by picture, 25 60

H

horizontal text, formatting of, 47

I

I, integer format, 32 60
Inserting a constant text, 51
Insertion of commas with *C*, 38
Insufficient space for the result in exponential form, 17
Integer format *I*, 32

K

K, scale factor qualifier, 36 60

L

L, left justification qualifier, 36 61
Leading and trailing zeros with thorn, 13
Leading and trailing zeros in picture format, 27
Leading decorators combined with leading zeros, 44
leading sign-dependent decorators, 62
leading zero, substitute for, 39
leading zeros, summary of rules regarding *Z* qualifier, 61
Leading zeros with *Z*, 37
"left argument" of a format phrase, 24
left argument of □*FMT*, 20
Left justifying with qualifier *L*, 36 61
list, to format several arrays at same time, 19
Literal text within a format phrase, 22

M

M, leading decorator for negative values, 42 62
marks, dividing, inserted by use of substitute for field overflow, 41

meaning and structure of control pairs in left argument of ⍕, 58
Monadic thorn applied to character array, 10
More digits than the system can supply, 7 18
Moving back over the fields already written, 54
Moving to a new position in the display with phrases *X* and *T*, 53

N

N, trailing decorator for negative values, 42 62
names, as examples of horizontal text to be formatted, 47
negative sign, space for, 13
position of, in picture format, 27
substitute for, when used as sign of exponent, 39
substitute decoration for, 42
negative values, different alignment for, 44
new position in the display, moving to, 53
Number of control pairs required by dyadic thorn, 11
numeric data formatted with character code, 50

O

One control pair applied to all columns, 12
Other qualifiers or decorators with *A* format, absence of, 50
overflow, field, when data cannot be represented in field provided, 6 18 29 32
Overwriting one field with another, 55

P

P, leading decorator for non-negative values, 42 62
pair, control one applied to all columns, 12
pairs, control, number of, 11
parentheses to indicate negative value, 43
phrase, format, list of, 3
separation of one from another, 21
format, repeated, within format control vector, 21
format, structure of, 23 59
format, last used, rules regarding, 59
summary of components of, 60
Picking an appropriate field width for decimal displays, 14
picture format, 25 60
leading and trailing zeros with, 27
sign-dependent decorators with, 28
floating dollar sign with, 28
trailing zeros and blanked-out field for zero value, 29
field overflow with, 29
background text with, 30
substitution of control characters with, 31
Position of negative sign or decorators in picture format, 28
precision and field width, how controlled in dyadic use of thorn, 10
predictable size and shape, of result of format primitives, 5
protection, cheque, characters for, 46

Q

Q, trailing decorator for non-negative values, 42 62
⎕*FMT*, detailed description of, 19
⎕*OUT* files, special property of ⎕*FMT* with, 3
⎕*PW* does not affect result of monadic thorn, 9
qualifiers, summary of, 24 25 60
Qualifiers and decorators, list of, 24
Qualifiers and decorators and the formats with which they are available, 24 25 60
Qualifiers usable with *I*, *F*, and *E* formats, 35
qualifiers with *A* format, absence of, 50

R

R, decorator to provide background text, 45 59
relative reposition, 53
Repeated phrases, 21 59
reposition, relative and absolute, 53
Representing numbers in exponential form, 15
"Right argument" of a format phrase, 23
Right argument of ⎕*FMT*, 19
rows, different numbers of, in arrays formatted together with ⎕*FMT*, 20

S

S, substitute-character code, 39 63
Scale factor *K*, 36
scaling qualifier, use with picture format to establish decimal point, 26
Selecting decimal or exponential form, 11
semicolons, as separators in right argument of ⎕*FMT*, 19
Separating one phrase from the next, 21
several arrays, shapes of, when formatted together, 19 20

severay arrays, formatted together with □*FMT*, 19
shape and conformability rules, summary of, 57
shape and size, predictable, in result of format primitives, 5
Shape of the arrays to be formatted, 19
sign, negative, space for, 13
negative, substitute decoration for, 42
Sign-dependent decorators in picture format, 28
Sign-dependent decorators *M*, *N*, *P*, and *Q*, 42
Sign-dependent decorators with *E* format, 44
sign-dependent decorators, 62
significant digits, with monadic thorn, controlled by □*PP*, 8
Single system-picked width for all columns, 13
size and shape, predictable, in result of format primitives, 5
Skipping to the end of the positions already formatted, 56
Space for the negative sign, 13
Structure and meaning of control pairs in the left argument of ⍕, 58
Structure of □*FMT* control vector, 59
Structure of a format phrase, 23
Substitute characters to insert dividing marks, 41
Substitutes for standard characters in *I*, *F*, and *E* formats, 39
Substituting other characters for 9 or *Z* in the picture, or for 0 or * in the result, picture format, 31
Substitution of standard characters, summary of rules regarding, 63
summaries of formatting rules, 57
system-picked width, 12
System-picked width in exponential form, 17
system-picked width for all columns, single, 13
system display, captured with monadic thorn, 8

T

T, absolute reposition, 53
with no stated position moves to end of fields already formatted, 56
text, literal, within a format phrase, 22
background, for unused positions within a field, 45
background, for zero values, 46
background, between print positions used for character data, 50
constant, insertion of, 51 63
constant, following last data column, 51
literal, summary of rules regarding, 59
text delimiters, 21 59
thorn, detailed description of, 8
dyadic, summary of rules regarding left argument of, 58
too small, field, 6
trailing sign-dependent decorators, 62
trailing zeros, with thorn, 14
Trailing zeros and blanked-out field for zero value, in picture format, 29
trailing zeros in picture format, 27
type, opposite, effect of trying to format, 50 60
Types of "left argument" to a format phrase, 24
Types of □*FMT* formatting phrases, 2

U

Using substitute characters to insert dividing marks, 41

V

vector, treated as one-column matrix by □*FMT*, 20

W

What the format primitives are for, 1
wider than needed, field, 6
width, single system-picked, for all columns, 13
system-picked, in exponential form, 17
Width of fields in result of monadic ⍕, 9
work area, special property of □*FMT* with, 3

X

X, relative reposition, 53

Z

Z, leading-zero qualifier, 37 61
zero field width, as indicator of system-picked width with dyadic thorn, 12
with □*FMT*, 52
zero value, represented by blank, 37
background text to mark, 46
represented by blank field, 61
zeros, leading, 37
leading, combined with leading decorators, 44

ISBN 0 86493 028 3